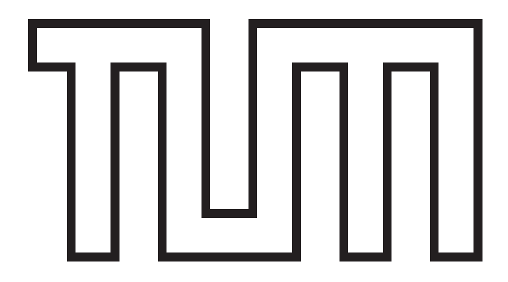


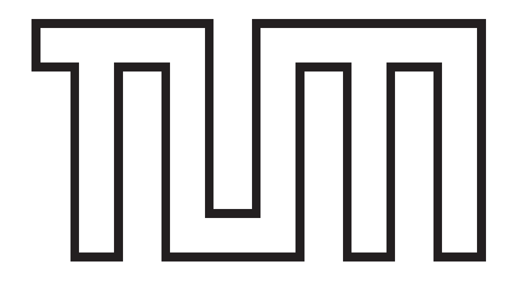
Technische Universität München

Fakultät für Informatik

Möglichkeiten plattformunabhängiger Klangsynthese

Interdisziplinäres Projekt

Matthias Thar
Sebastian Gutsfeld



Technische Universität München

Fakultät für Informatik

Möglichkeiten plattformunabhängiger Klangsynthese

Interdisziplinäres Projekt

Matthias Thar
Sebastian Gutsfeld

Themensteller: Christian Böhm

Betreuer: Christian Böhm

Abgabetermin: 7. März 2006

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Cabel

Cabel ist eine graphische Benutzeroberfläche zum Entwerfen, Erstellen und Spielen von Csound5-Instrumenten. Wie bei modularen Synthesizern entstehen Cabel/Csound-Instrumente durch das Verbinden von Modulen über Patchkabel.

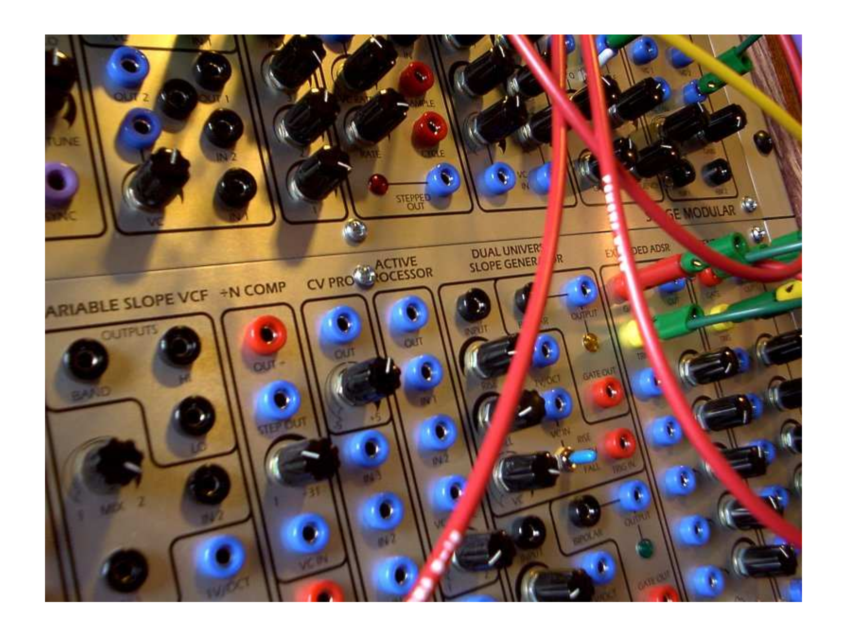


Abbildung 1.1: Analoger, modularer Synthesizer von Serge

Die Module, die Sie in Cabel verwenden, sind in XML[1] Dateien definiert und kapseln

[1]eXtensible Markup Language, ein Standard zur Erstellung von maschinen- und menschenlesbaren Dokumenten.

Csound Anweisungen zu kleinen Blackbox Einheiten, deren Ein- und Ausgänge untereinander verbunden werden können.

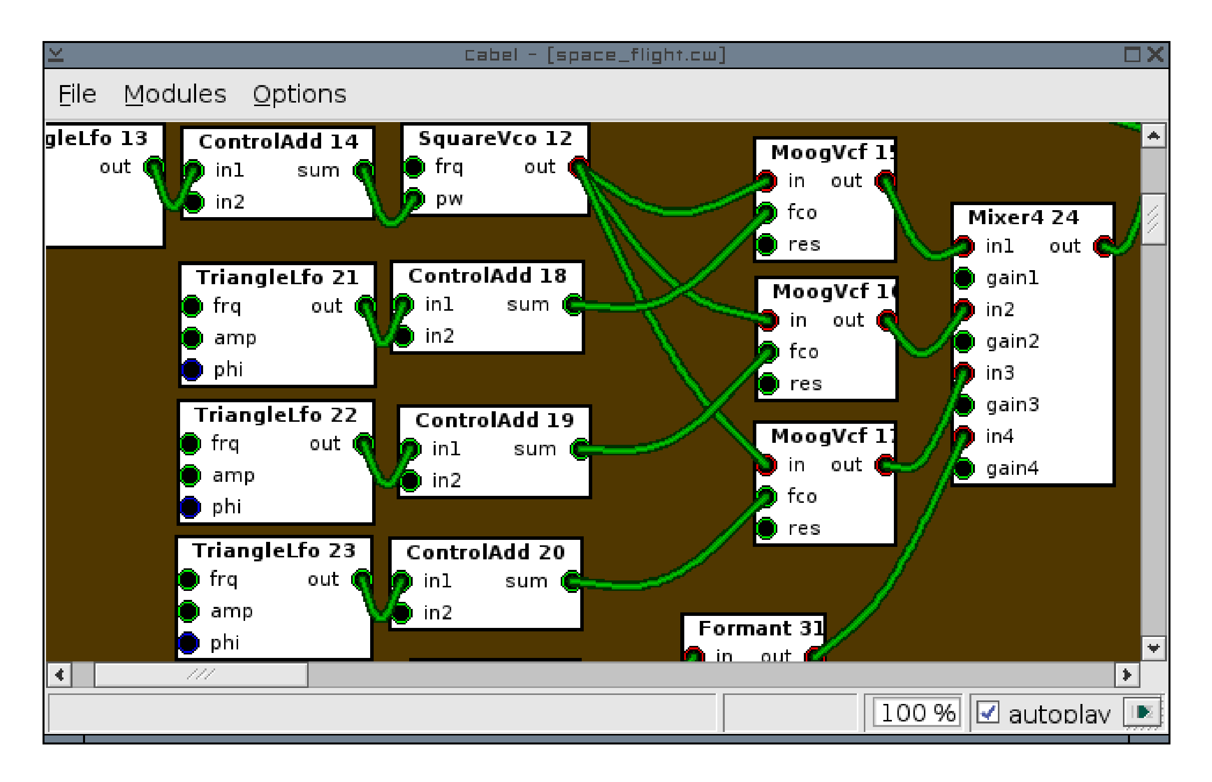


Abbildung 1.2: Verschiedene miteinander verbundene Module in Cabel

1.1.1 Zielgruppe und Anspruch

Der ursprüngliche Anspruch der Entwickler war, ein Werkzeug zu entwerfen, das sowohl Csound Anfänger als auch erfahrene Csounder anspricht und unabhängig vom verwendeten Betriebssystem läuft. Nähere Informationen zu Csound in Kapitel 1.2.1 auf S. 8.

Cabel soll den Csound Einsteigern ermöglichen, die Grundlagen dieser wunderbaren Sprache auf möglichst einfache und anschauliche Art und Weise kennenzulernen. Den anspruchsvolleren Anwendern soll Cabel als Werkzeug zum schnellen Testen von Ideen für neue Instrumente und Spielen mit deren Einstellungen anhand einer graphischen Benutzeroberfläche dienen.

1.1.2 Erweiterbarkeit

Dadurch, daß die in Cabel verwendeten Module in XML Dateien definiert sind, in die Csound Befehle eingebettet werden, ist die Erweiterbarkeit von Cabel nur durch die Möglichkeiten von Csound beschränkt und die sind wie in Abschnitt 1.2.1 kurz beschrieben fast grenzenlos.

Nähere Informationen zur Erweiterung von Cabel durch eigene Module entnehmen Sie bitte Kapitel 7 auf S. 56.

1.2 Arbeitsumgebung von Cabel

1.2.1 Was ist Csound5? - Eine kurze Einführung

Csound ist ein mächtiges und vielseitiges Software Synthesizer Programm, mit dem sich sowohl Instrumente als auch Effekte programmieren lassen.

Es generiert Musik, indem textbasierte, sogenannte Orchester Dateien (*.orc) in Csound Maschinencode übersetzt werden und an diesen Maschinencode einzelne Noten Events geschickt werden.

Der Quelltext in Orchester Dateien beschreibt Csound-Instrumente und ist im wesentlichen in 2 Blöcke, dem *Header*- und dem *Instrument-Block*, unterteilt.

Im Header werden die *Sample Rate* und *Control Rate*, in der die nachfolgenden Instrumente verarbeitet werden, definiert. Außerdem wird im Header die *Anzahl der Ausgabe-Kanäle* angegeben.

Im Instrument-Block wird dann durch Csound Befehle, die *Opcodes* genannt werden, das Instrument beschrieben. Diese Opcodes lassen sich über ihre Ein- und Ausgabe-Variablen miteinander verknüpfen.

Csound5 liefert eine Menge an Opcodes zum Generieren, Modifizieren, Einlesen und Ausgeben von Signalen. Außerdem gibt es die Möglichkeit externe VST[2] Effekte und Synthesizer einzubinden, Instrumente durch Skript Sprachen wie Python zu erweitern, etc.

Um die so programmierten Instrumente spielen zu können, braucht der Soundprozessor noch die Information, was er wann und wie spielen soll. Dies legt eine weitere Eingabequelle fest, die Noten Events und dazu gehörige Parameter an den Prozessor übermittelt.

Csound kann dabei folgende Noteneingabequellen verstehen:

- Echtzeit-MIDI[3]
- MIDI-Dateien
- OSC[4] Nachrichten
- Score Dateien (von Csound vorgegebenes Format)

Weiterführende Links zu Csound finden Sie im Anhang B auf S. 149.

[2] Erkärung zu VST unter http://de.wikipedia.org/wiki/Virtual_Studio_Technology
[3] Erklärung zu MIDI unter http://de.wikipedia.org/wiki/MIDI
[4] Open Sound Control http://www.cnmat.berkeley.edu/OpenSoundControl/

1.2.2 Python

Ein weiterer Aspekt, der Cabel sehr vielseitig einsetzbar und leicht erweiterbar macht, ist die Programmiersprache Python, in der die Anwendung entwickelt wurde.

Python[5] ist eine interpretierte, interaktive, objektorientierte Programmiersprache, die mit dem Ziel entworfen wurde möglichst einfach und übersichtlich zu sein

Da Cabel nicht nur in Python programmiert wurde, sondern auch eine Python Shell integriert hat (Abbildung 1.3), kann der fortgeschritten Anwender bestimmte Aufgaben mit Hilfe kleiner Programme automatisieren. Wie man auf die Funktionalität von Cabel über Python zugreifen kann wird in Kapitel 8 auf S. 62 genauer beschrieben.

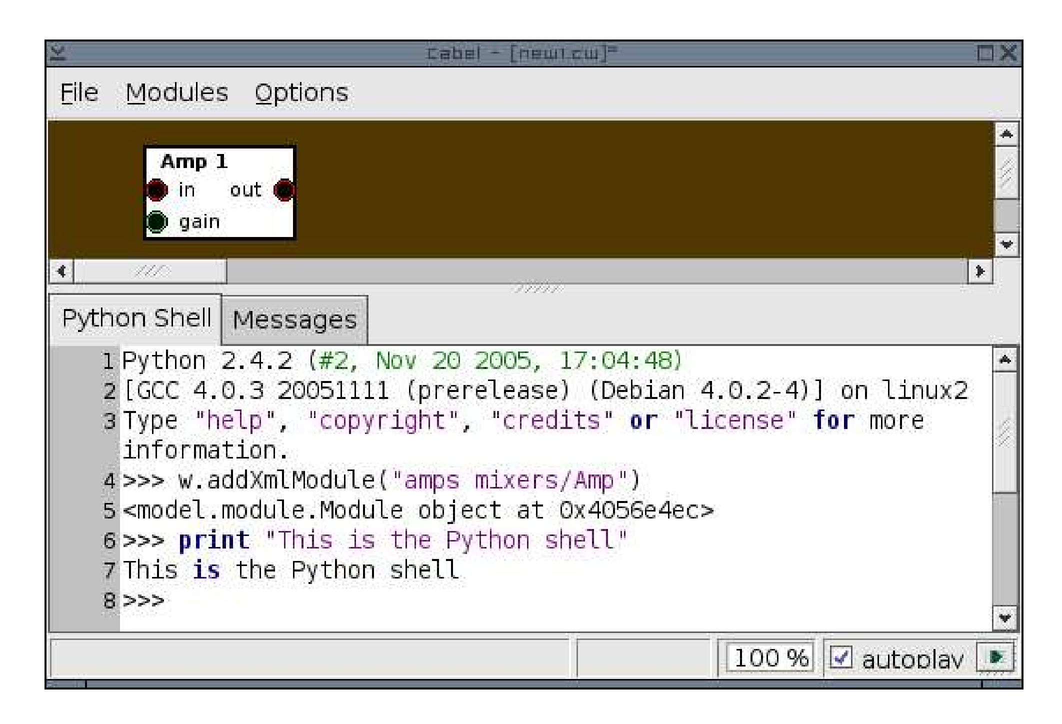


Abbildung 1.3: Die in Cabel integrierte Python Shell

1.3 Neuigkeiten und Support

Auf der offiziellen Cabel Homepage unter `http://cabel.sourceforge.net` erhalten Sie Updates und wichtige Neuigkeiten zu Cabel.

Auf der Sourceforge Projekt Seite von Cabel unter `http://sourceforge.net/projects/cabel` haben Sie, falls Sie Probleme mit Ihrer Cabel Installation haben sollten, Zugriff auf Benutzerforen und die Möglichkeit Fehler zu melden oder Vorschläge für die Weiterentwicklung von Cabel zu machen.

[5] http://www.python.org

1.4 Kommentare von Cabel Anwendern

Congratulations! This is already a very neat program in it's first release.

I am looking at Cabel very much as it is well done.

(Steven Yi - Entwickler der Csound Kompositionsumgebung blue[6])

Great job; well done.

I have played (a little) with Cabel and it seems easy and intuitive, with a nice interface. It ran straight out of the box on windows, which has a lot to be said of.

(Victor Lazzarini - Leiter des Music Technology Laboratory der National University of Ireland, Maynooth)

Excellent work! It is truly an amazing idea to have a dataflow version of the Csound language.

I feel that we are at the dawn of a new age of accessibility with Csound because of tools such as blue Composition Environment and Cabel modular patching system.

The line between any proprietary implementation and Csound is blurring more every day; from a UI perspective, namely. Thanks to your contributions!

(David Akbari - Musiker)

it gives me the "tweakiness" of Reaktor or PD with the power of Csound.

(Michael Gogins - Csound Entwickler und Komponist)

I am thrilled to see that the program is written in cross-platform Python and look forward to having an opportunity to see how well it works on MacOS 9.

Bravo!

(Anthony Kozar - Csound Entwickler und Komponist)

[6]http://www.csounds.com/stevenyi/blue/index.html

Kapitel 2

Installation

Cabel basiert auf der Programmiersprache Python. Um Python Programme ausführen zu können, benötigt man die darin verwendeten Bibliotheken und einen Python Interpreter, der das Programm ausführt.

Die graphische Benutzeroberfläche für Cabel wurde mit der wxWidgets Implementierung für Python entwickelt. wxWidgets ist ein GUI[1] Toolkit, eine Sammlung von Bausteinen zur Generierung von Benutzeroberflächen.

Um Cabel ausführen zu können, muß also folgende Software installiert sein:

- Die Programmiersprache Python (`http://www.python.org`),
- Das GUI Toolkit wxPython (`http://www.wxpython.org`) und
- Csound5 (`http://www.csounds.com`).

Auf die Installation dieser für ein funktionierendes Cabel essentiellen Programme wird im Folgenden genauer eingegangen.

2.1 Windows Installationsanleitung

2.1.1 Python

2.1.1.1 Python 2.4



Unter `http://www.python.org` können Sie das Installationsprogramm für Python 2.4 (oder höher) herunterladen. Starten Sie den Installer und folgen Sie den Anweisungen.

Hinweis zur benötigten Version von Python:

[1] Graphical User Interface

Um Cabel mit älteren Versionen von Python zu verwenden, muß die Installation von Csound5 angepaßt werden, da aktuelle Releases von Csound5 für Windows mit Python 2.4 übersetzt wurden.

2.1.1.2 Python für Windows Extensions

Um Cabel auch in Echtzeit verwenden zu können, wird Csound5 aus Cabel heraus automatisch in einem extra Prozess gestartet. Damit dies in Windows reibungslos funktioniert, wird eine Erweiterung der Standard Python Bibliotheken, die Python for Windows Extensions, benötigt.



Unter `http://sourceforge.net/projects/pywin32` können Sie das für Ihre Python Version entsprechende Installationsprogramm für die Extensions herunterladen. Starten Sie den Installer und folgen Sie den Anweisungen.

2.1.1.3 wxPython



Unter `http://www.wxpython.org/download.php#binaries` können Sie das für Ihre Python Version entsprechende Installationsprogramm für die wxWidgets (Version 2.5.0 oder höher) herunterladen. Starten Sie den Installer und folgen Sie den Anweisungen.

2.1.2 Csound5



Unter `http://sourceforge.net/projects/csound` können Sie das Installationsprogramm des aktuellen Csound5 Releases für Windows herunterladen. Starten Sie den Installer und folgen Sie den Anweisungen der Installationsroutine.

2.1.3 Cabel

Jetzt fehlt nur noch Cabel.



Unter `http://sourceforge.net/projects/cabel` kann das aktuelle Cabel Release heruntergeladen werden. Dort findet man ein Zip-Archiv, das alle Cabel Dateien enthält.

Dieses Archiv muß in ein Verzeichnis, wie z.B. `C:\Cabel\`, entpackt werden.

Hinweis zur Aktualität des Codes:

Cabel ist Open Source Software und wird weiterhin Veränderungen unterliegen und weiterentwickelt werden. Da nicht jede dieser Veränderungen unmittelbar einen Release zur Folge haben wird, kann es durchaus interessant sein, sich anstatt des letzten Releases den aktuellen Stand von Cabel aus dem ebenfalls auf Sourceforge gehosteten CVS zu holen.

Für nähere Informationen hierzu sei auf die Rubrik *C*VS auf der Homepage von Cabel auf sourceforge.net[2] verwiesen: `http://sourceforge.net/projects/cabel`.

2.1.4 Setzen von Umgebungsvariablen

Damit die Programme, die Cabel benötigt, voneinander wissen, müssen noch die folgenden Umgebungsvariablen gesetzt werden:

1. Der Umgebungsvariable `PYTHONPATH` das Unterverzeichnis `bin` ihres Csound5 Installationsverzeichnisses hinzufügen.

   Beispiel: Ist Csound5 im Verzeichnis `C:\Csound5\` installiert, so muß die Umgebungsvariable `PYTHONPATH` in seiner durch Srichpunkte unterteilten Liste von Verzeichnissen, den Wert `C:\Csound5\bin` enthalten.

2. Die Umgebungsvariable `PATH` enthält ebenfalls eine durch Strichpunkte unterteile Liste von Verzeichnissen. Dieser Liste sollte noch der Pfad zum Python 2.4 Interpreter (`Python.exe`) hinzugefügt werden.

   Beispiel: Ist Python 2.4 im Verzeichnis `C:\Python24\` installiert, so muß genau dieser Wert in der Umgebungsvariablen `PATH` aufgelistet sein.

Setzen von Umgebungsvariablen unter Windows:

Dazu mit der rechten Maustaste auf das *A*rbeitsplatz- Symbol auf dem Desktop oder im Explorer gehen und *E*igenschaften auswählen. Dann den Reiter *E*rweitert auswählen und den Button *U*mgebungsvariablen... betätigen. Zum Bearbeiten einer vorhandenen Umgebungsvariable diese im Bereich *S*ystemvariablen auswählen und den Button *B*earbeiten... betätigen, oder zur Neuanlage auf *N*eu....

2.2 Linux Installationsanleitung

2.2.1 Python

2.2.1.1 Python 2.4

Falls Sie eine aktuelle Linux Distribution verwenden, sollte Python 2.4 bereits als fertiges Paket enthalten sein. Ansonsten müssen Sie die aktuelle Version unter `http://www.python.org` herunterladen und der Anleitung zur Installation folgen.



[2] siehe Kapitel 1.3, S. 9.

2.2.1.2 wxPython



wxPython sollte ebenfalls in einer aktuellen Linux Distrubution enthalten sein. Ansonsten müssen sie wxPython selber kompilieren. Die entsprechenden Quellpakete inklusive Installationsanleitung finden Sie unter `http://www.wxpython.org/download.php#sources`

2.2.2 Csound5



Unter `http://csound.sourceforge.net` kann man ein aktuelles Csound5 herunterladen. Hier werden sowohl fertig kompilierte Versionen als RPM Paket als auch der Csound5 Quellcode zum Selberkompilieren angeboten. Dabei sei auf die in Csound5 enthaltene Installationsanleitung verwiesen.

2.2.3 Cabel

Jetzt fehlt nur noch Cabel.



Unter `http://sourceforge.net/projects/cabel` kann das aktuelle Cabel Release heruntergeladen werden. Dort findet man ein Tar-Gz-Archiv, das alle Cabel Dateien enthält.

Hinweis zur Aktualität des Codes:

Cabel ist Open Source Software und wird weiterhin Veränderungen unterliegen und weiterentwickelt werden. Da nicht jede dieser Veränderungen unmittelbar einen Release zur Folge haben wird, kann es durchaus interessant sein, sich anstatt des letzten Releases den aktuellen Stand von Cabel aus dem ebenfalls auf Sourceforge gehosteten CVS zu holen.

Für nähere Informationen hierzu sei auf die Rubrik CVS auf der Homepage von Cabel auf sourceforge.net[3] verwiesen: `http://sourceforge.net/projects/cabel`.

2.3 Starten von Cabel

Nach der Installation kann Cabel gestartet werden. Öffnen sie dazu unter Windows die MS-DOS Eingabeaufforderung, wechseln in das Verzeichnis Ihrer Cabel Installation (z.B. mit `cd C:\Cabel`) und starten Sie Cabel mit dem Befehl `python cabel.py`.

Standardmäßig werden Dateien mit der Endung *.py* mit Ihrem installierten Python Interpreter verknüpft, sodaß es ausreicht Cabel mit einem Doppelklick auf *cabel.py* zu starten.

Das Starten von Cabel unter Linux erfolgt analog mit dem Befehl `python cabel.py`.

[3] siehe Kapitel 1.3, S. 9.

Kapitel 3

Einstellungen im Options Menü

Um Cabel sinnvoll einsetzen zu können, müssen Sie die Einstellungen von Cabel an Ihr System anpassen. Dazu erklären wir im folgenden Kapitel die Einträge des *Options* Menüs (Abbildung 3.1).

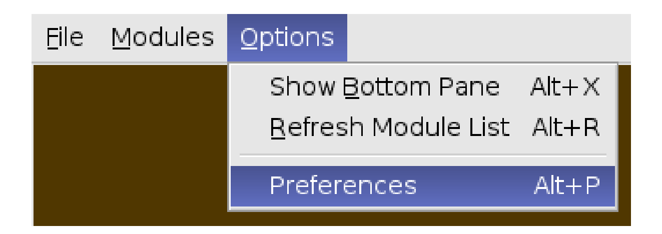


Abbildung 3.1: Das Options Menü

3.1 Ein- und Ausblenden des unteren Fensters

Das untere Fenster kann über das Menü *Options→Show Bottom Pane* oder durch die Tastenkombination `ALT-X` ein- bzw. ausgeblendet werden.

Es besteht aus den folgenden Reitern:

a) **Python Shell**
Das ist ein Python Interpreter, über den Zugriff auf die gesamte API[1] Cabels besteht. Was einem das bringt und wie diese Schnittstelle anzusprechen ist erfahren Sie in Kapitel 8 ab Seite 62.

b) **Messages**
In diesem Reiter werden Fehlermeldungen und sonstige Meldungen, die Cabel auf die Standardausgabe schreibt, ausgegeben.

[1] Application Programming Interface

3.2 Aktualisieren der Liste von Cabel Modulen

Haben Sie Cabel XML Module verändert oder neu in das Modul Verzeichnis[2] hinzugefügt[3] und wollen diese in Cabel nutzen, so muß die Cabel-interne Liste an Modulen durch den Aufruf von *Options→Refresh Module List* im Menü oder durch die Tastenkombination `ALT-R` aktualisiert werden.

3.3 Cabel Konfigurator

Der Cabel Konfigurationsdialog wird durch die Tastenkombination `ALT-P` oder über das Menü *Options→Preferences* geöffnet.

3.3.1 Csound Einstellungen

Im Reiter *Csound* (Abbildung 3.2) läßt sich Csound für die Verwendung mit Cabel konfigurieren.

Der Reiter ist unterteilt in die 3 Blöcke *Instrument Header*, *Score* und *Csound Preferences*:

1. Instrument Header

- **Sample Rate**
  Samples, die pro Sekunde und Kanal berechnet werden. Dies sollte der internen Sample Rate Ihrer Soundkarte entsprechen.
  Voreingestellter Wert: 44100.
  Csound Handbuch: `http://www.csounds.com/manual/html/sr.html`.
- **Control Rate**
  Rate, wieviele Kontroll-Signale pro Sekunde verarbeitet werden.
  Voreingestellter Wert: 4410.
  Csound Handbuch: `http://www.csounds.com/manual/html/kr.html`.
- **KSMPS**
  Anzahl der Samples in einer Kontroll-Zeitspanne. Der Wert dieses Parameters muß dem Bruch $\frac{SampleRate}{ControlRate}$ entsprechen.
  Voreingestellter Wert: 10.
  Csound Handbuch: `http://www.csounds.com/manual/html/ksmps.html`.

[2]Siehe Kapitel 3.3.3 auf Seite 22
[3]Auf das *Schreiben eigener Module* wird in Kapitel 7 ab Seite 56 genauer eingegangen

Abbildung 3.2: Der Reiter *Csound* im Konfigurations Dialog

- **Channels**
  Anzahl der Ausgabe Kanäle (1 = Mono, 2 = Stereo, 4 = Quadrophonisch).
  Csound Handbuch: `http://www.csounds.com/manual/html/OrchHeader.html`.

2. Score

Hier wird die *Score*[4] für Csound angegeben.

Um das Cabel Instrument[5] mit der Nummer `1` von der Startzeit 0 an automatisch 60

[4]engl.: Partitur
[5]Nähere Informationenen zu Instrumenten in Kapitel 4.2.2, S. 29 und Kapitel 4.3, S. 30

Sekunden lang spielen zu lassen, genügt die Score Zeile

```
i1 0 60
```

Entsprechend ändern Sie den `i1` Eintrag zu `i2`, wenn Sie Instrument 2 automatisch triggern wollen.

Wollen Sie Ihr Instrument durch ein MIDI Gerät triggern und steuern, so erledigt das die Score Zeile

```
f0 6000
```

Dieses *f-statement*[6] läßt Csound im Hintergrund laufen, ohne ein Cabel Instrument zu starten. Ein Cabel Instrument kann so durch ein MIDI Note-On Signal mit der Kanal-Nummer, die der Instrumentnummer entspricht, getriggert werden.

Wie in Kapitel 1.2.1 (S.8) erwähnt, kann als Noten Eingabe für Csound– und damit auch für Cabel– auch eine Score Datei verwendet werden. Weitergehende Informationen über Csound Scores finden Sie im Csound Benutzerhandbuch[7].

3. Csound Preferences

- **Csound Path**
  Pfadangabe des Csound Compilers `csound.exe`. Dieser ist normalerweise im Unterverzeichnis `bin` Ihrer Csound Installation zu finden.
- **Csound Parameters**
  Mit den Csound Parametern kann man u.a. einstellen, woher Csound welche Eingaben, wie z.B. MIDI, oder Echtzeit Audio nimmt, und wie die daraus generierten Sounds ausgegeben werden.

  Windows Beispiele:

  * Für MIDI Eingabe und Echtzeit Audio Ausgabe über den Audio-Treiber MME:
    `-d -b128 -B1536 -W -+rtmidi=mme -M0 -+rtaudio=mme -o dac1`
  * Für Echteit Audio Input aus dem MME Gerät 0 den Parametern `-i adc0` hinzufügen.

  Linux Beispiele:

  * Für Echtzeit Audio Ausgabe über den ALSA Treiber:
    `-d -W -o dac:plughw:0 -+rtaudio=alsa -b256 -B2048 -M0 -m0`

[6] Ein *f-statement* definiert eine Funktionstabelle, die eine Folge von Werten repräsentiert.
[7] `http://www.csounds.com/manual`

  * Für Echtzeit Audio Ausgabe über den JACK Soundserver:
    `-d -W -o dac -+rtaudio=jack -b256 -B2048 -M0`

  * Für Echtzeit Audio Input mit JACK den Parametern `-i adc` hinzufügen, für ALSA `-i adc:plughw:0`.

  Eine ausführliche Beschreibung aller möglichen Csound Kommandozeilenparameter gibt es unter `http://www.csounds.com/manual/html/CommandFlags.html`

- **Compilation Feedback Timeout**
  Cabel startet den Csound Prozessor in einem separaten, asynchron zu Cabel laufenden Prozess. Bisher lassen sich der aufrufende Cabel- und der aufgerufene Csound-Prozess nicht plattformunabhängig und performant synchronisieren. Der Parameter *Compilation Feedback Timeout* ist Teil eines Workarounds, der versucht, das Synchronisierungsproblem zu lösen:
  Damit Cabel weiß, ob ein abzuspielendes Instrument vom Csound Prozess erfolgreich übersetzt wurde, wartet es nach dem Starten die in *Compilation Feedback Timeout* in Millisekunden angegebene Zeitspanne ab und überprüft erst dann, ob der Csound Prozess noch läuft. Ist dies nicht der Fall bedeutet das, daß Csound einen Fehler entdeckt hat und Cabel wechselt wieder zurück in den Zustand *Gestoppt*.

  Da die Zeit, die Csound zum Übersetzen der Cabel Instrumente benötigt, von der Größe des Cabel Instruments und der Geschwindigkeit Ihres Rechners abhängt, können Sie diesen Parameter hier selbst einstellen. Dabei gehen Sie wie folgt vor:

  1. Provozieren Sie für ein zu übersetzendes Instrument einen Übersetzungsfehler, indem Sie z.B. einen unsinnigen *Channel* Wert (z.B. 2 für ein Mono-Instrument) für den *Instrument Header* einstellen.

  2. Starten Sie den Csound Prozess durch Betätigen des *Start Csound* Knopfs mit dem *Play Symbol* rechts unten im Cabel Fenster[8].

  3. Warten Sie, bis der Csound Prozess in der Console, aus der Sie Csound gestartet haben, das Abbrechen durch die Ausgabe einer Fehlermeldung bestätigt.

     a) Wenn der Start/Stop Csound Knopf rechts unten im Cabel Fenster jetzt immer noch das *Play Symbol* anzeigt, so ist der Wert korrekt eingestellt, da Cabel das Abbrechen des Csound Prozesses registriert hat und sich nicht fälschlicherweise im Zustand *Gestartet* befindet.

     b) Zeigt der Start/Stop Csound Knopf das *Stop Symbol*, so betätigen Sie ihn erneut, um Cabel in den Zustand *Gestoppt* zu bringen, erhöhen den *Compilation Feedback Timeout* Wert und wiederholen den Vorgang ab Punkt 2.

[8] Das Starten, bzw. Stoppen des Csound Prozesses über den Csound Start/Stop Knopf in der Statusleiste wird in Kapitel 4.3.3 auf Seite 30 genauer erläutert

Autoplay

Zusätzlich zu diesen drei Blöcken können Sie die *Autoplay* Funktion von Cabel aktivieren. Ist *Autoplay* aktiv und läuft Csound im Hintergrund, wird jedesmal, wenn Sie die Parameter eines Moduls ändern, der aktuelle Csound Prozess beendet und ein neuer gestartet.

Deaktivieren Sie *Autoplay*, so müssen Sie nach jeder Parameteränderung an den Modulen Csound von Hand beenden und neu starten, um Ihre Änderungen zu hören.

3.3.2 Anpassen der Benutzeroberfläche

Der Reiter *User Interface* (Abbildung 3.3) ermöglicht es Ihnen, das Erscheinungbild, aber auch das Verhalten der Benutzeroberfläche von Cabel an Ihre Bedürfnisse anzupassen.

1. Workspace

Der Workspace ist die eigentliche Arbeitsfläche von Cabel, auf der Sie Module platzieren und miteinander verknüpfen.

In der Konfigurationsgruppe *Workspace* können Sie:

- die Hintergrundfarbe (*Workspace Colour*) der Cabel Arbeitsflächein in einem Farbauswahldialog, wie in Abbildung 3.4 dargestellt, verändern und
- die Größe (*Workspace Width/Height*) des scrollbaren Bereichs der Arbeitsfläche festlegen.

2. Full Module Names

Durch das Aktivieren des Kontrollkästchens *Full Module Names* wird in den Cabel Modulen der relative Pfad (ausgehend vom *Modul Verzeichnis*[9]) der entsprechenden XML Modul Datei angezeigt. Ansonsten ist nur der Modulname ohne relativen Pfad zu sehen.

3. Warning On Removing Modules

Das Kontrollkästchen aktiviert/deaktivert die Ausgabe von Warnungen beim Löschen von Cabel Modulen von der Arbeitsfläche.

[9]siehe Kapitel 3.3.3.

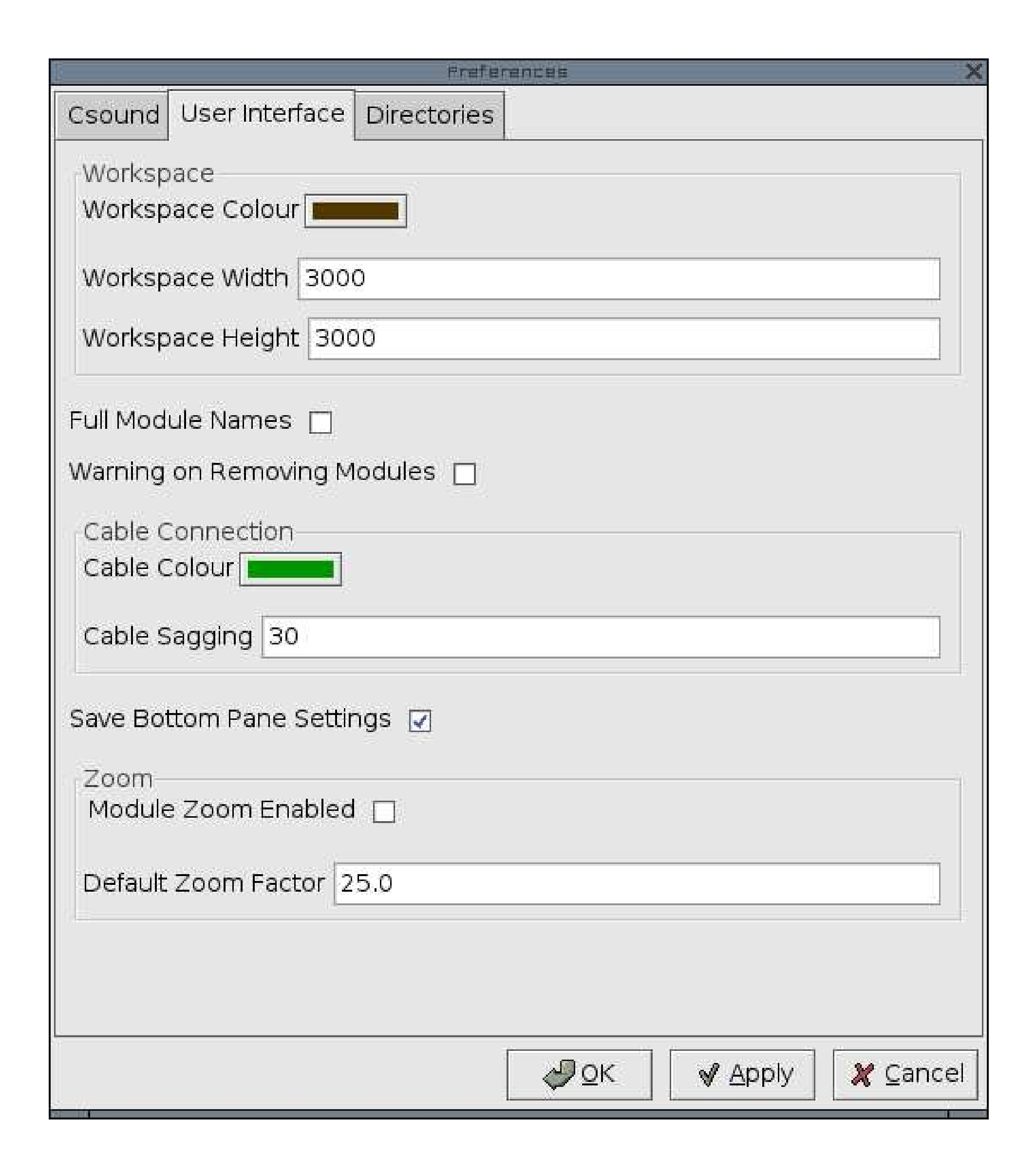


Abbildung 3.3: Der Reiter *User Interface* im Konfigurations Dialog

4. Cable Connection

In der Konfigurationsgruppe *Cable Connection* lassen sich die Anzeige Eigenschaften der Patchkabel verändern:

- Die Farbe der Patchkabel (*Cable Colour*) kann in einem Farbauswahldialog, wie in Abbildung 3.4 dargestellt, ausgewählt werden.

- Der Grad, in dem die Patchkabel auf der Arbeitsfläche "durchhängen", wird durch den Wert *Cable Sagging* eingestellt.

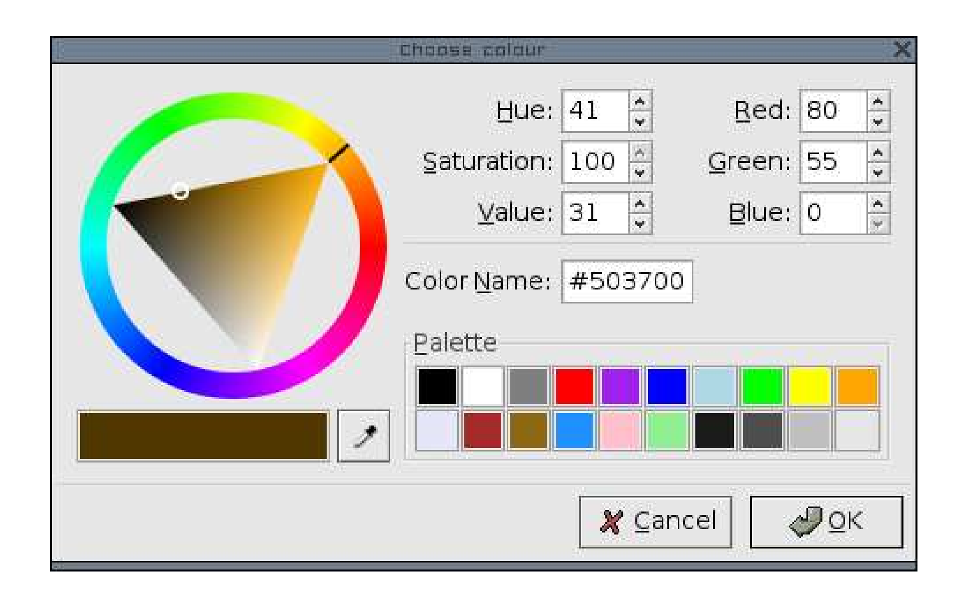


Abbildung 3.4: Der Farbauswahldialog von Cabel

5. Save Bottom Pane Settings

Das Kontrollkästchen *Save Bottom Pane Setting* legt fest, ob Sichtbarkeit und ausgewählter Reiter des unteren Fensters[10] (Python Shell und Messages) gespeichert werden.

6. Zoom

In der Konfigurationsgruppe *Zoom* kann das Zoomen von einzelnen Modulen aktiviert, bzw. deaktiviert werden (Kontrollkästchen *Module Zoom Enabled*) und der vorbelegte Vergrößerungs- bzw. Verkleinerungs-Faktor in Prozent (*Default Zoom Factor*) angegeben werden.

Ist diese Option aktiviert, so können einzelne Module in deren Kontextmenü[11] vergrößert, bzw. verkleinert werden.

3.3.3 Verzeichnisse

Im Reiter *Directories* (Abbildung 3.5) werden die für Cabel relevanten Verzeichnisse angegeben.

[10]siehe Kapitel 3.1.
[11]siehe Kapitel 4.2, S. 27.

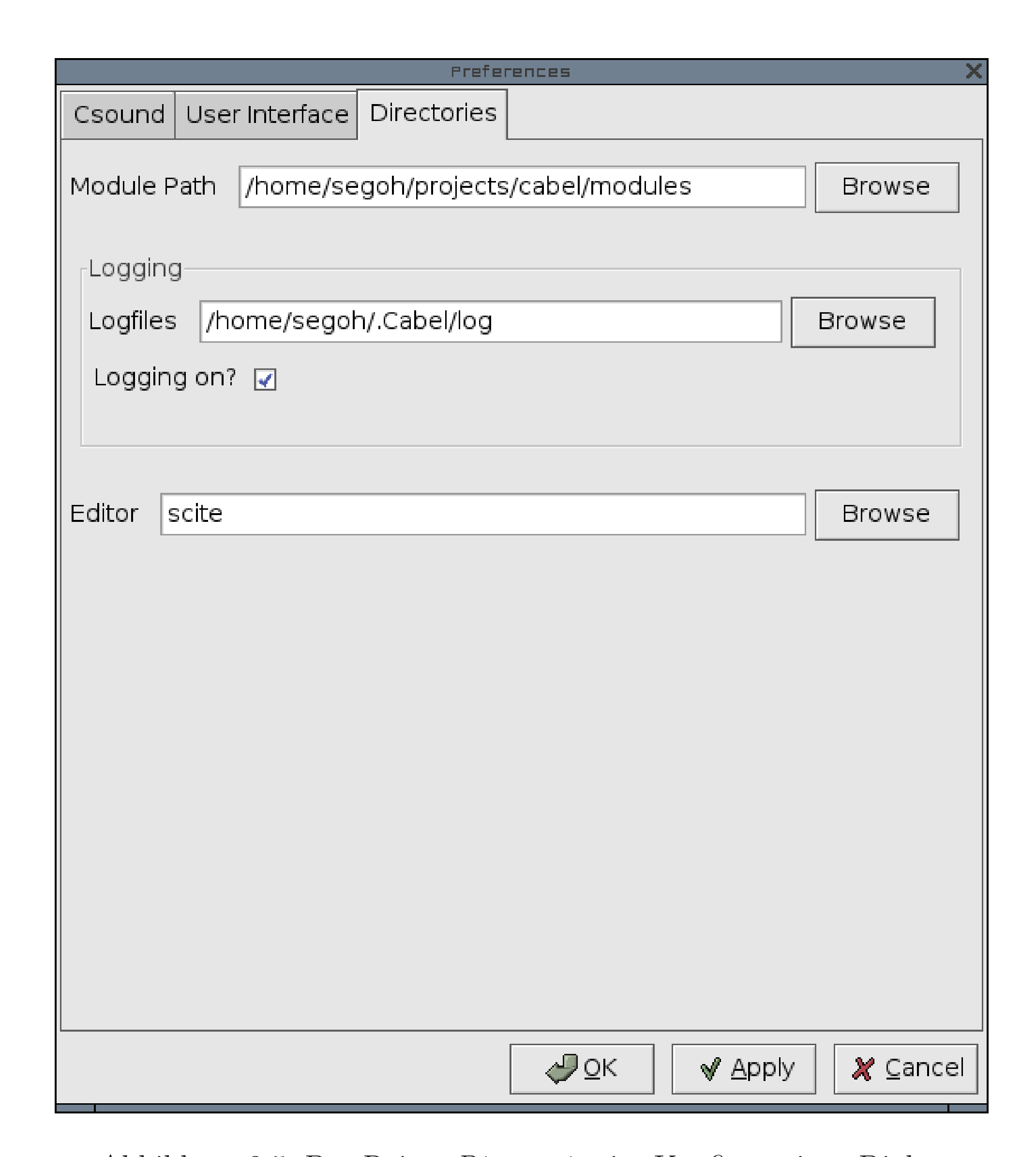


Abbildung 3.5: Der Reiter *Directories* im Konfigurations Dialog

1. Module Path

Hier wird das Verzeichnis, das die XML Modul Dateien enthält, festgelegt.

Diese XML Dateien können auch in Unterverzeichnissen organisiert werden. Cabel erzeugt aus jedem Unterverzeichnis ein extra Untermenü im Modules Menü[12], sodaß Sie Ihre Module nach eigenen Vorstellungen gruppieren können.

[12]siehe Abbildung 4.2, S. 26.

2. Logging

Die Konfigurationsgruppe *Logging* legt fest, ob die Standardausgabe[13] von Cabel in Dateien mitgelogged wird (*Logging on?*), und wenn ja, in welchem Verzeichnis diese Log-Dateien abgelegt werden.

3. Editor

Hier kann der Texteditor angegeben werden, den Cabel ausführen soll, wenn Sie z.B. über das Modul-Kontextmenü[14] den Quellcode eines Moduls angezeigt bekommen wollen.

[13]siehe Kapitel 3.1.
[14]siehe Kapitel 4.2, S. 27.

Kapitel 4

Benutzeroberfläche

Die Benutzeroberfläche von Cabel ist in drei große Bereiche aufgeteilt, die im folgenden Kapitel näher beschrieben werden.

4.1 Menü

4.1.1 File

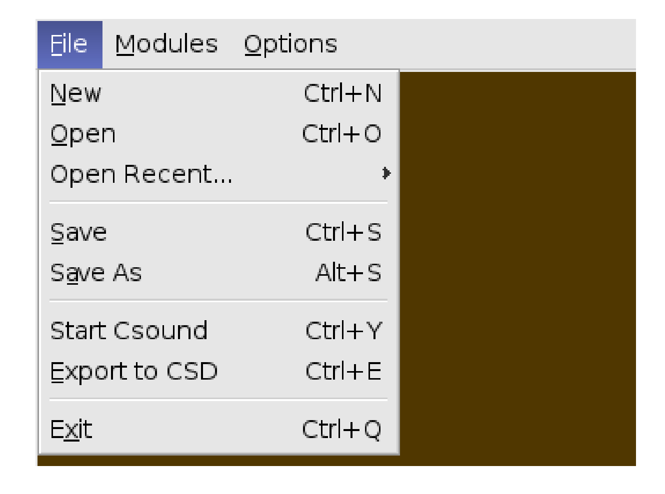


Abbildung 4.1: Das File Menü

Das File Menü läßt Sie Instrumente neu erzeugen, laden, speichern, starten, exportieren und das Programm beenden.

New Löscht den Arbeitsplatz und läßt Sie ein neues Instrument erzeugen.

Open Läßt Sie gespeicherte Instrumente öffnen.

Open Recent Liste der zehn zuletzt geöffneten Cabel Instrumente.

Save Speichert die aktuellen Instrumente unter dem aktuellen Namen.

Save As Speichert die aktuellen Instrumente unter einem neuen Namen.

Start Csound Startet die aktuell auf Ihrem Arbeitsplatz befindlichen Instrumente mit Csound. Läuft bereits ein Csound Prozess, kann mit diesem Menüeintrag Csound gestoppt werden (siehe auch Kapitel 4.3.3 auf S. 30)

Export to CSD Exportiert Ihren Arbeitsplatz in eine Textdatei, die Sie ohne Cabel direkt mit Csound starten können[1].

Exit Beendet Cabel.

4.1.2 Modules

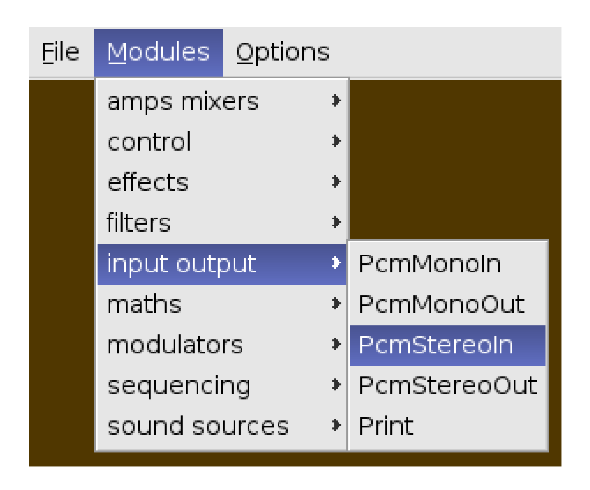


Abbildung 4.2: Das Modules Menü

In diesem Menü können Sie die Module auswählen, die Sie dem Arbeitsplatz hinzufügen wollen. Alternativ erreichen Sie dieses Menü wenn Sie mit der rechten Maustaste auf einen freien Bereich des Cabel Arbeitsplatzes klicken.

4.1.3 Options

Dieses Menü wurde bereits in Kapitel 3 auf S. 15 näher beschrieben.

[1]Unified File Format von Csound: `http://www.csounds.com/manual/html/CommandUnifile.html`

4.2 Arbeitsplatz

Der Arbeitsplatz nimmt den größten Teil der Benutzeroberfläche in Anspruch. Auf ihm befinden sich die Module und deren Verbindungen.

Um den sichtbaren Ausschnitt des Arbeitsplatzes zu verschieben, können Sie entweder die Scrollleisten verwenden oder den Arbeitsplatz mit gedrückter mittlerer Maustaste verschieben.

4.2.1 Module

Um ein neues Modul zu erzeugen, wählen Sie das Gewünschte aus dem Modulmenü (Abbildung 4.2). Dieses Menü erreichen Sie ebenfalls über einen Klick mit der rechten Maustaste auf einen freien Bereich des Arbeitsplatzes.

Um ein Modul wieder vom Arbeitsplatz zu entfernen, öffnen Sie das Kontextmenü (Abbildung 4.3) des Moduls mit einem Klick der rechten Maustaste auf das entsprechende Modul und wählen den Punkt *Remove Module* aus. Mit *Show Module Xml* können Sie sich den mit diesem Modul verknüpften XML Quelltext in einem Editor Ihrer Wahl (Kapitel 3.3.3) anzeigen lassen.

Falls Sie wie in Kapitel 3.3.2 beschrieben *Module Zoom Enabled* aktiviert haben, können Sie im Modul Kontextmenü das Modul in *Default Zoom Factor* Schritten vergrößern und verkleinern.

Wenn Sie die linke Maustaste über einem Modul gedrückt halten, können Sie es auf der Arbeitsfläche platzieren.

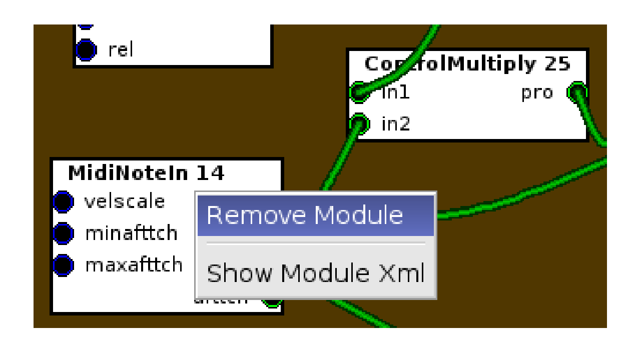


Abbildung 4.3: Kontextmenü eines Moduls

4.2.1.1 Aufbau

Wie in Abbildung 4.4 zu sehen besteht ein Modul aus seinem Namen, einer intern zugeordneten ID Nummer, Eingängen auf der linken Seite und Ausgängen auf der rechten Seite.

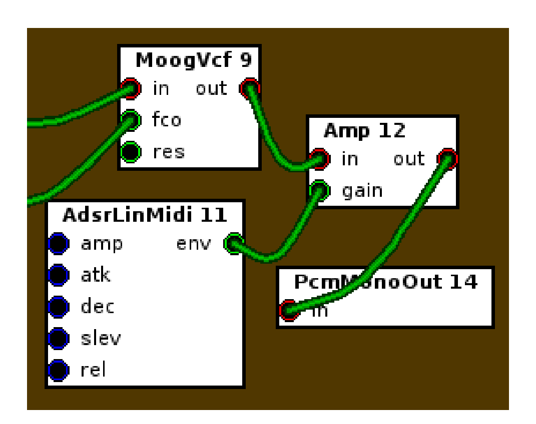


Abbildung 4.4: Miteinander verbundene Module

Die Farbe der Ein- und Ausgänge signalisiert, wie Csound die eingestellten oder verbundenen Werte intern verarbeitet.

rot Audio Signal, das zur eingestellten *Audio Rate* (Kapitel 3.3.1) aktualisiert wird.

grün Kontroll Signal, das zur eingestellten *Control Rate* (Kapitel 3.3.1) aktualisiert wird.

blau Instrument Signal, das nur aktualisiert wird, wenn das Instrument neu getriggert wird.

Da Module keine internen Parameter besitzen, werden sämtliche Einstellungen über die Eingänge eingestellt, entweder direkt über das Wertefenster (Kapitel 4.2.1.2) oder, indem die Module untereinander verbunden werden (Kapitel 4.2.1.3).

4.2.1.2 Das Wertefenster

Mit einem Doppelklick auf ein Modul öffnet sich das zugehörige Wertefenster (Abbildung 4.5), in dem Sie sämtliche Parameter des Moduls einstellen können. Die Wertzuweisung kann sowohl über den Schieberegler als auch über die Texteingabe des entsprechenden Parameters erfolgen. Bleiben Sie mit Ihrem Mauszeiger kurz über dem Textfeld oder dem Schieberegler eines Parameters, erscheint ein kurzer Hilfetext, der nähere Informationen und den erlaubten Wertebereich des Parameters auflistet.

Über die Tastaturkombination `CTRL-W` können Sie das Wertefenster wieder schließen.

Wie in Abbildung 4.5 bei Parameter *in* zu sehen, werden Schieberegler und Texteingabefeld eines Parameters im Wertefenster deaktiviert, wenn dieser verbunden ist und seinen Wert über den Ausgang eines anderen Moduls erhält.

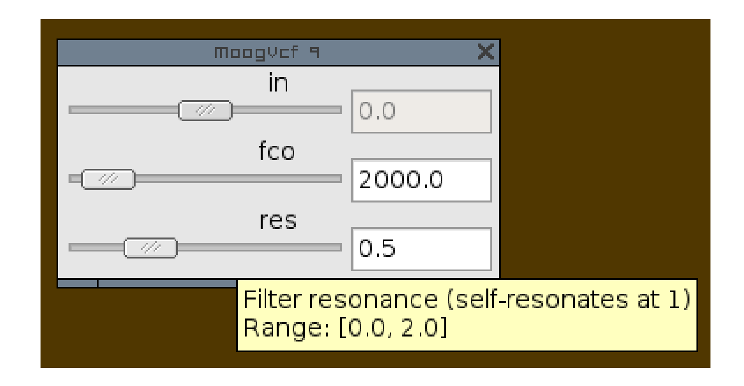


Abbildung 4.5: MoogVcf Wertefenster mit verbundenem *in* Eingang und Tool Tipp zu dem Parameter *res*

4.2.1.3 Verbinden von Modulen

Damit ein Eingangsparameter den Wert eines Ausgangs eines anderen Moduls zugewiesen bekommt, müssen Sie beide verbinden. Befindet sich Ihr Mauszeiger über einem Ausgang, verwandelt er sich in eine Hand. Wenn Sie nun die linke Maustaste drücken und bei gedrückter Maustaste bewegen, entsteht zwischen dem ausgewählten Ausgang und Ihrem Mauszeiger ein Kabel. Bewegen Sie den Mauszeiger mit dem Kabel über den Eingang eines anderen Moduls, verwandelt er sich wieder in eine Hand. Wenn Sie die linke Maustaste jetzt wieder loslassen, einsteht eine Verbindung zwischen den beiden Modulen.

Ebenso können sie die Verbindung zweier Module wieder "ausstecken", indem sie die Verbindung mit gedrückter linker Maustaste aus dem Eingang des zugehörigen Moduls "ziehen".

Wenn Sie zwei Ausgänge unterschiedlicher Art, also z.B. ein Audio Signal mit einem Kontroll Signal, verbinden, wird das Signal auf die geringere Rate der beiden Signale konvertiert.

4.2.2 Instrumente

Eine Gruppe von verbundenen Modulen nennt man *Instrument*. Jedem *Instrument* ist eine eindeutige Nummer zugeordnet, die in der Statusleiste (siehe Kapitel 4.3 auf Seite 30) angezeigt wird.

Falls Sie Ihr Instrument über MIDI spielen wollen, entspricht die Instrumentnummer dem MIDI Kanal, über den Sie Ihr Instrument ansprechen können. Nähere Informationen wie Instrumenten MIDI Kanäle zugewiesen werden entnehmen Sie bitte dem Csound Benutzerhandbuch unter `http://www.csounds.com/manual`.

4.3 Statusleiste

Die Statusleiste (Abbildung 4.6) zeigt zusätzliche Informationen zu Menüpunkten, Modulen, Moduleingängen und Modulausgängen, über denen sich der Mauszeiger gerade befindet.



Abbildung 4.6: Die Statusleiste zeigt Informationen zu einem *Amp* Modul an

Weiterhin wird in der Statusleiste die Instrumentnummer des Moduls angezeigt (Kapitel 4.2.2), über dem sich der Mauszeiger befindet.

4.3.1 Zoomfaktor

In dem Texteingabefeld der Statusleiste können Sie den Zoomfaktor der Module auf Ihrem Arbeitsplatz einstellen, um Modul- und Schriftgröße innerhalb der Module gemäß Ihren Vorstellungen anzupassen.

Mit der Tastenkombination `CTRL-Z` springen Sie direkt in die Zoomfaktor Texteingabe, um den Wert zu ändern.

4.3.2 Autoplay

Mit diesem Schalter können Sie die *Autoplay* Funktion an- und ausschalten. Nähere Informationen zur *Autoplay* Funktion erhalten Sie auf S. 20 in Kapitel 3.3.1.

4.3.3 Play und Stop Knopf

Zeigt dieser Knopf (zu sehen unten rechts in Abbildung 4.6) ein dreieckiges *Play* Zeichen, läuft momentan kein Csound Prozess, und Sie können durch Betätigen des Knopfes Csound mit den Instrumenten auf Ihrem Arbeitsplatz starten.

Zeigt dieser Knopf ein rechteckiges *Stop* Zeichen, läuft aktuell ein Csound Prozess, den Sie durch Betätigen des Knopfes beenden können.

Damit die Statusanzeige diese Knopfes richtig funktioniert, muß das *Compilation Feedback Timeout* richtig eingestellt sein (Kapitel 3.3.1 auf S. 19).

Sollte es beim Start Ihrer Instrumente zu Fehlern kommen, erhalten Sie die Csound Fehlermeldungen in dem Fenster der Eingabeaufforderung, in dem Sie Cabel gestartet haben.

Der Knopf läßt sich auch mit der Tastenkombination `CTRL-Y` betätigen.

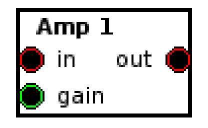
Kapitel 5

Module

Dieses Kapitel gibt einen Überblick über eine kleine Auswahl der mitgelieferten Standardmodule von Cabel.

5.1 Amps Mixers

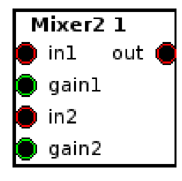
5.1.1 Amp



Verstärker für ein Audio Signal. Das Eingangssignal wird um den Gain Faktor[1] verstärkt bzw. abgeschwächt.

in	Eingang für ein Audio Signal
gain	Faktor der Verstärkung
out	Audio Ausgang für das verstärkte Signal

5.1.2 Mixer2



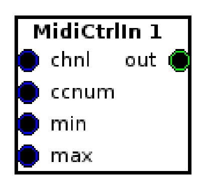
[1]In Kapitel 6.1.1 auf Seite 47 wird näher auf den Verstärkungsfaktor bei Cabel Instrumenten eingegangen.

Mischer für zwei Audio Signale. Die beiden Eingangssignale werden um den jeweiligen Gain Faktor[2] verstärkt bzw. abgeschwächt und aufsummiert, um das Ausgabesignal zu erzeugen.

in1	Eingang für das erste Audio Signal
gain1	Faktor der Verstärkung des ersten Audio Signals
in2	Eingang für das zweite Audio Signal
gain2	Faktor der Verstärkung des zweiten Audio Signals
out	Audio Ausgang für das gemischte Signal

5.2 Control

5.2.1 MidiCtrlIn



Input für MIDI Controller Änderungen. Dieses Modul empfängt einkommende MIDI Controller Änderungen und skaliert sie auf den mit *min* und *max* eingestellten Bereich.

chnl	MIDI Kanal
ccnum	MIDI CC Nummer
min	Minimaler Wert, auf den skaliert werden soll
max	Maximaler Wert, auf den skaliert werden soll
out	Skalierter Wert der ankommenden MIDI Controller Änderung

5.2.2 MidiNoteIn



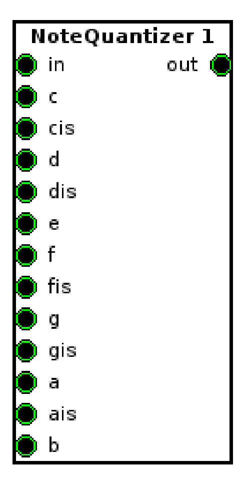
Input für MIDI Note On, Note Off, Velocity On und Channel Aftertouch Events. Für ankommende MIDI Note On Signale wird sowohl die entsprechende MIDI Notennummer

[2]In Kapitel 6.1.1 auf Seite 47 wird näher auf den Verstärkungsfaktor bei Cabel Instrumenten eingegangen.

als auch die zugehörige Frequenz ausgegeben. Der zugehörige MIDI Kanal entspricht dabei der Instrumentennummer, die in Cabels Statusleiste angezeigt wird. MIDI Velocity On des ankommenden Events wird ebenfalls ausgewertet und auf den Bereich 0 bis *velscale* skaliert. Ebenso werden Channel Aftertouch Events empfangen und skaliert ausgegeben.

velscale	Maximaler Wert, auf den Velocity On Werte skaliert werden
minafttch	Minimaler Wert, auf den Aftertouch skaliert werden soll
maxafttch	Maximaler Wert, auf den Aftertouch skaliert werden soll
frq	Ausgabe der Frequenz eines MIDI Note On Events
note	Ausgabe der MIDI Notenummer eines MIDI Note On Events
vel	Skalierter MIDI Velocity On Wert
afttch	Skalierter Channel Aftertouch Wert

5.2.3 NoteQuantizer



MIDI Noten Quantisierer. NoteQuantizer rundet das ankommende Kontroll Signal, sodaß nur MIDI Notenummern der eingestellten Noten ausgegeben werden. Die Augabe von NoteQuantizer kann mit Hilfe des *Midi2Frq* Moduls in die entsprechende Frequenz umgerechnet werden.

in	Eingang für ein Kontroll Signal
c	Schalter für die Note C
cis	Schalter für die Note C#
d	Schalter für die Note D
dis	Schalter für die Note D#
e	Schalter für die Note E
f	Schalter für die Note F
fis	Schalter für die Note F#
g	Schalter für die Note G
gis	Schalter für die Note G#
a	Schalter für die Note A
ais	Schalter für die Note A#
b	Schalter für die Note H
out	Quantisierte MIDI Notennummer

5.2.4 SlewLimiter

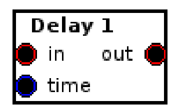


Abstiegsbegrenzer / Portamento Controller. Mit SlewLimiter wird ein linearer Übergang zwischen zwei aufeinanderfolgenden Werten des Eingangssignals über die eingestellte Zeitspanne erzeugt. Dadurch können abrupte Parametersprünge geglättet werden.

in	Eingang für ein Kontroll Signal
time	Anstiegs-/Abstiegszeit in Sekunden
out	Begrenztes Kontroll Signal

5.3 Effects

5.3.1 Delay

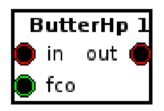


Delay für Audio Signale. Das ankommende Audio Signal wird um *time* Sekunden verzögert ausgegeben.

in	Eingang für ein Audio Signal
time	Verzögerungszeit in Sekunden
out	Verzögertes Audio Signal

5.4 Filters

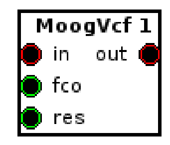
5.4.1 ButterHp



Butterworth Hochpass Filter. Dieses Filter ist eine Implementierung eines Butterworth Hochpass Filters zweiter Ordnung mit einer Flankensteilheit von 12 dB pro Oktave. Die Module *ButterBp*, *ButterBr* und *ButterLp* sind die entsprechenden Bandpass, Kerbfilter und Tiefpass Versionen dieses Filters.

in	Eingang für ein Audio Signal
fco	Cutoff Frequenz
out	Gefiltertes Audio Signal

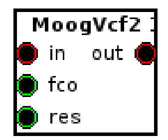
5.4.2 MoogVcf



Emulation eines Moog Tiefpass Filters. Dieses Filter ist eine digitale Emulation der Moog Diodenleiter Konfiguration. Das Filter gerät bei einer Resonanz von etwa 1 in Selbstoszillation. Wegen der zugrunde liegenden Csound Implementierung dieses Filters ist es wichtig, daß das Eingangssignal eine maximale Amplitude von 1 hat, da es ansonsten zu Clipping kommt.

in	Eingang für ein Audio Signal (mit maximaler Amplitude von 1)
fco	Cutoff Frequenz
res	Resonanz
out	Gefiltertes Audio Signal

5.4.3 MoogVcf2



Emulation eines Moog Tiefpass Filters. Im Gegensatz zu *MoogVcf* kann dieses Modul auch Audio Signale mit einer größeren Amplitude als 1 filtern. Dafür kann die Resonanz nicht bis zur Selbstoszillation erhöht werden, da es ansonsten zu extremen Verzerrungen kommt.

in	Eingang für ein Audio Signal
fco	Cutoff Frequenz
res	Resonanz
out	Gefiltertes Audio Signal

5.5 Input Output

5.5.1 PcmMonoIn



Eingang für ein Mono Signal von der Soundkarte.

out	Ausgang für das ankommende Audio Signal

5.5.2 PcmMonoOut



Ausgabe des ankommenden Mono Signals an die Soundkarte. Achten Sie darauf, daß bei Verwendung dieses Moduls der Parameter *Channels* im Csound Options Dialog (zu erreichen über das Menü *Options*→*Preferences*) auf 1 gesetzt ist.

in	Eingang für ein Audio Signal

5.5.3 PcmStereoIn



Eingang für ein Stereo Signal von der Soundkarte.

left	Ausgang für das ankommende Audio Signal des linken Kanals
right	Ausgang für das ankommende Audio Signal des rechten Kanals

5.5.4 PcmStereoOut

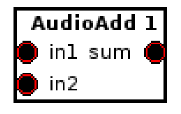


Ausgabe des ankommenden Stereo Signals an die Soundkarte. Achten Sie darauf, daß bei Verwendung dieses Moduls der Parameter *Channels* im Csound Options Dialog (zu erreichen über das Menü *Options*→*Preferences*) auf 2 gesetzt ist.

in_l	Audio Signal für linken Stereokanal der Soundkarte
in_r	Audio Signal für rechten Stereokanal der Soundkarte

5.6 Maths

5.6.1 AudioAdd



Addiert die beiden ankommenden Audio Signale.

in1	Erstes Audio Signal
in2	Zweites Audio Signal
sum	Summe der beiden ankommenden Audio Signale

5.6.2 AudioMultiply



Multipliziert die beiden ankommenden Audio Signale.

in1	Erstes Audio Signal
in2	Zweites Audio Signal
pro	Produkt der beiden ankommenden Audio Signale

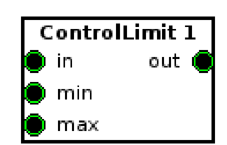
5.6.3 ControlAdd



Addiert die beiden ankommenden Kontroll Signale.

in1	Erstes Kontroll Signal
in2	Zweites Kontroll Signal
sum	Summe der beiden ankommenden Kontroll Signale

5.6.4 ControlLimit



Beschränkt das ankommende Kontroll Signal auf einen bestimmten Wertebereich.

in	Eingang für Kontroll Signal
min	Untere Grenze des Wertebereichs, auf den *in* beschränkt werden soll
max	Obere Grenze des Wertebereichs, auf den *in* beschränkt werden soll
out	Auf den Wertebereich beschränktes Kontroll Signal

5.6.5 ControlMultiply



Multipliziert die beiden ankommenden Kontroll Signale.

in1	Erstes Kontroll Signal
in2	Zweites Kontroll Signal
pro	Produkt der beiden ankommenden Kontroll Signale

5.6.6 Midi2Frq

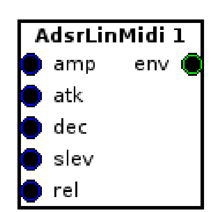


Rechnet MIDI Notennummern in die entsprechende Frequenz um.

in	MIDI Notenummer
out	Ausgabe der entsprechenden Frequenz

5.7 Modulators

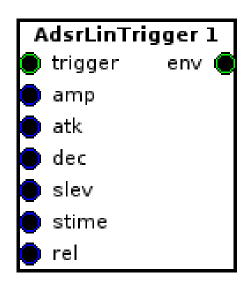
5.7.1 AdsrLinMidi



MIDI getriggerter linearer ADSR Hüllkurvengenerator. Dieser Hüllkurvengenerator erzeugt eine klassische lineare ADSR Hüllkurve (Attack, Decay, Sustain, Release). AdsrLinMidi wird durch ein ankommendes MIDI Note On Signal getriggert.

amp	Amplitude, auf die die normalisierte Hüllkurve skaliert werden soll
atk	Attack Zeit in Sekunden
del	Delay Zeit in Sekunden
slev	Normalisierter Sustain Level
rel	Release Zeit in Sekunden
env	Hüllkurve als Kontroll Signal

5.7.2 AdsrLinTrigger



Kontroll Signal getriggerter linearer ADSR Hüllkurvengenerator. Dieser Hüllkurvengenerator erzeugt eine klassische lineare ADSR Hüllkurve (Attack, Decay, Sustain, Release). AdsrLinTrigger wird durch ein ankommendes Trigger Signal gestartet. Dieses Trigger Signal kann jedes Signal sein, das vom Wert 0 auf einen Wert gleich 1 oder höher springt. Als Trigger Signal bietet sich ein *PulseLfo* Modul an.

trigger	Trigger Kontroll Signal
amp	Amplitude, auf die die normalisierte Hüllkurve skaliert werden soll
atk	Attack Zeit in Sekunden
del	Delay Zeit in Sekunden
slev	Normalisierter Sustain Level
stime	Sustain Zeit in Sekunden
rel	Release Zeit in Sekunden
env	Hüllkurve als Kontroll Signal

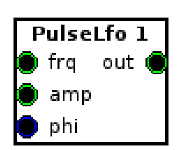
5.7.3 EnvFollower



Hüllkurvenfolger. EnvFollower gibt die Amplitude des ankommenden Audio Signals als Kontroll Signal aus.

in	Audio Signal Eingang
out	Amplitude des ankommenden Audio Signals

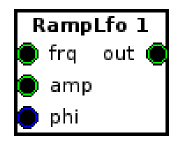
5.7.4 PulseLfo



Pulswellen Niederfrequenz-Oszillator. Dieser Niederfrequenz-Oszillator erzeugt einen periodischen kurzen positiven Impuls mit der eingestellten Amplitude. Dadurch eignet sich PulseLfo ideal als Trigger Signal für andere Module.

frq	Frequenz
amp	Amplitude
phi	Startphase des Oszillator Signals
out	Ausgabe des Oszillator Signals als Kontroll Signal

5.7.5 RampLfo



Invertierter Sägezahnwellen Niederfrequenz-Oszillator. Dieser Niederfrequenz-Oszillator erzeugt eine periodische invertierte Sägezahnwelle mit der eingestellten Amplitude.

frq	Frequenz
amp	Amplitude
phi	Startphase des Oszillator Signals
out	Ausgabe des Oszillator Signals als Kontroll Signal

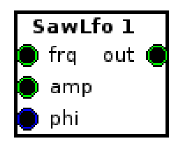
5.7.6 SampleAndHold



Abtast- und Halte-Modul. SampleAndHold erzeugt aus dem ankommenden Signal ein treppenförmiges Kontroll Signal. Dabei wird das ankommende Kontroll Signal bei jedem Trigger Impuls abgetastet (sample) und bis zum nächsten Trigger Signal am Ausgang gehalten. Das Trigger Signal kann jedes Signal sein, das vom Wert 0 auf einen Wert gleich 1 oder höher springt. Als Trigger Signal bietet sich ein *PulseLfo* Modul an.

trigger	Trigger Kontroll Signal
in	Ankommendes Kontroll Signal
out	Treppenförmiges Kontroll Signal

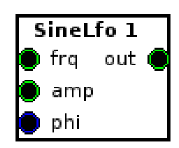
5.7.7 SawLfo



Sägezahnwellen Niederfrequenz-Oszillator. Dieser Niederfrequenz-Oszillator erzeugt eine periodische Sägezahnwelle mit der eingestellten Amplitude.

frq	Frequenz
amp	Amplitude
phi	Startphase des Oszillator Signals
out	Ausgabe des Oszillator Signals als Kontroll Signal

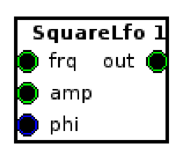
5.7.8 SineLfo



Sinuswellen Niederfrequenz-Oszillator. Dieser Niederfrequenz-Oszillator erzeugt eine periodische Sinuswelle mit der eingestellten Amplitude.

frq	Frequenz
amp	Amplitude
phi	Startphase des Oszillator Signals
out	Ausgabe des Oszillator Signals als Kontroll Signal

5.7.9 SquareLfo



Rechteckwellen Niederfrequenz-Oszillator. Dieser Niederfrequenz-Oszillator erzeugt eine periodische Rechteckwelle (symmetrische Pulswelle) mit der eingestellten Amplitude.

frq	Frequenz
amp	Amplitude
phi	Startphase des Oszillator Signals
out	Ausgabe des Oszillator Signals als Kontroll Signal

5.7.10 TriangleLfo

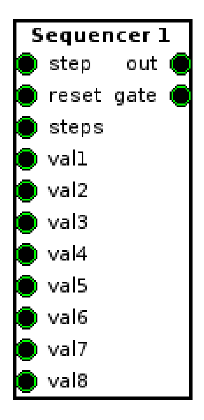


Dreieckwellen Niederfrequenz-Oszillator. Dieser Niederfrequenz-Oszillator erzeugt eine periodische Dreieckwelle mit der eingestellten Amplitude.

frq	Frequenz
amp	Amplitude
phi	Startphase des Oszillator Signals
out	Ausgabe des Oszillator Signals als Kontroll Signal

5.8 Sequencing

5.8.1 Sequencer



Acht Schritt Sequencer. Bei jedem ankommenden Trigger Signal gibt Sequencer den nächsten seiner Werte aus und springt beim letzten wieder zurück auf den ersten. Bei einem ankommenden Reset Signal springt der Sequencer ebenfalls auf den ersten Wert zurück. Zusätzlich wird ein Gate Signal ausgegeben, das 0 ist, falls der aktuell ausgegebene Wert 0 ist, und 1 für alle anderen Werte. Das Trigger und Reset Signal kann jedes Signal sein, das vom Wert 0 auf einen Wert gleich 1 oder höher springt. Als Trigger Signal bietet sich ein *PulseLfo* Modul an.

step	Trigger Signal / Clock Signal
reset	Reset Signal (springt auf den ersten Wert zurück)
steps	Anzahl der Schritte bis ein automatischer Reset erfolgt
val1	Erster Wert
val2	Zweiter Wert
val3	Dritter Wert
val4	Vierter Wert
val5	Fünfter Wert
val6	Sechster Wert
val7	Siebter Wert
val8	Achter Wert
out	Ausgabe des aktuellen Werts
gate	Gate Signal

5.9 Sound Sources

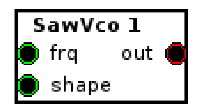
5.9.1 Noise



Rauschgenerator. Noise erzeugt weißes Rauschen, also einen Zufallswert zur Audio Rate mit der eingestellten Amplitude.

amp	Amplitude des Rauschens
out	Weißes Rauschen

5.9.2 SawVco



Sägezahn Oszillator. SawVco erzeugt eine periodische Sägezahnschwingung mit normalisierter Amplitude (=1). Die Wellenform kann mit dem *shape* Parameter stufenlos von invertierter Sägezahnwelle über Dreieckwelle bis hin zur "normalen" Sägezahnwelle eingestellt werden.

frq	Frequenz
shape	Wellenform (invertierter Sägezahn, Dreieck, Sägezahn)
out	Audio Signal des Sägezahn Oszillators

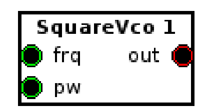
5.9.3 SineVco



Sinus Oszillator. SineVco erzeugt eine periodische Sinusschwingung mit normalisierter Amplitude (=1).

frq	Frequenz
out	Audio Signal des Sinus Oszillators

5.9.4 SquareVco



Pulswellen Oszillator. SquareVco erzeugt eine periodische Pulswelle mit normalisierter Amplitude (=1). Die Pulsbreite kann mit dem *pw* Parameter stufenlos eingestellt werden.

frq	Frequenz
shape	Pulsbreite
out	Audio Signal des Pulswellen Oszillators

Kapitel 6

Beispiele

Dieses Kapitel zeigt Ihnen Schritt für Schritt, wie Sie ein eigenes Cabel Instrument erzeugen und nutzen können.

6.1 Einfacher subtraktiver Synthesizer

Die einzelnen Arbeitsschritte dieses Beispielinstruments sind im *examples* Unterverzeichnis Ihrer Cabel Installation als durchnummerierte Dateien enthalten. Somit werden wir nur auf die wichtigsten Änderungen der einzelnen Schritte eingehen und Sie können sich die genauen Einstellungen der Instrumente anhand der Beispieldateien ansehen.

6.1.1 Der Sägezahn Oszillator

Zunächst beginnen wir mit einem einfachen Instrument, das einen konstanten Ton mit einem Sägezahn Oszillator erzeugt.

Den Sägezahn Oszillator erhalten Sie, indem sie entweder auf das *Modules* Menü klicken oder mit der rechten Maustaste auf einen freien Bereich des Cabel Arbeitsplatzes klicken, um das *Modules* Kontextmenü zu öffnen. Wählen Sie im *sound sources* Unterverzeichnis das *SawVco* Modul aus.

Um ein Audio Signal an die Soundkarte schicken zu können, benötigen Sie noch einen Ausgang für Ihr Instrument. Wählen sie im Modulmenü Unterverzeichnis *input output* das Modul *PcmMonoOut* aus.

Wenn Sie jetzt den *out* Ausgang des Oszillators direkt per Drag&Drop mit dem *in* Eingang des Soundkartenmoduls verbinden und das Instrument mit dem Play Knopf in der Statusleiste starten würden, wäre kein Ton zu hören, weil der Oszillator nur ein Signal mit der Amplitude 1 erzeugt und damit viel zu leise ist.

Diese Amplitude 1 ist ein Csound-internes Maß für Lautstärke. Die maximal darstellbare Lautstärke von Csound beträgt 32768 und stellt eine programminterne Größe dar. Merken Sie sich einfach, daß meist eine Lautstärke von 20000 ausreicht um Clipping[1] zu vermeiden.

Sie müssen also den Oszillator verstärken um ihn hören zu können. Dazu benötigen Sie das *Amp* Modul im *amps mixers* Unterverzeichnis.

Verbinden Sie den *out* Ausgang des SawVco mit dem *in* Eingang des Amp Moduls und den *out* Ausgang des Amp Moduls mit dem *in* Eingang von PcmMonoOut.

Durch Doppelklicken auf das SawVco Modul öffnen Sie dessen Einstellungen. Setzen Sie die Frequenz auf einen passenden Wert, z.B. 200 Hz. Im Amp Modul setzen Sie die Verstärkung, also den *gain* Parameter, auf 25000.

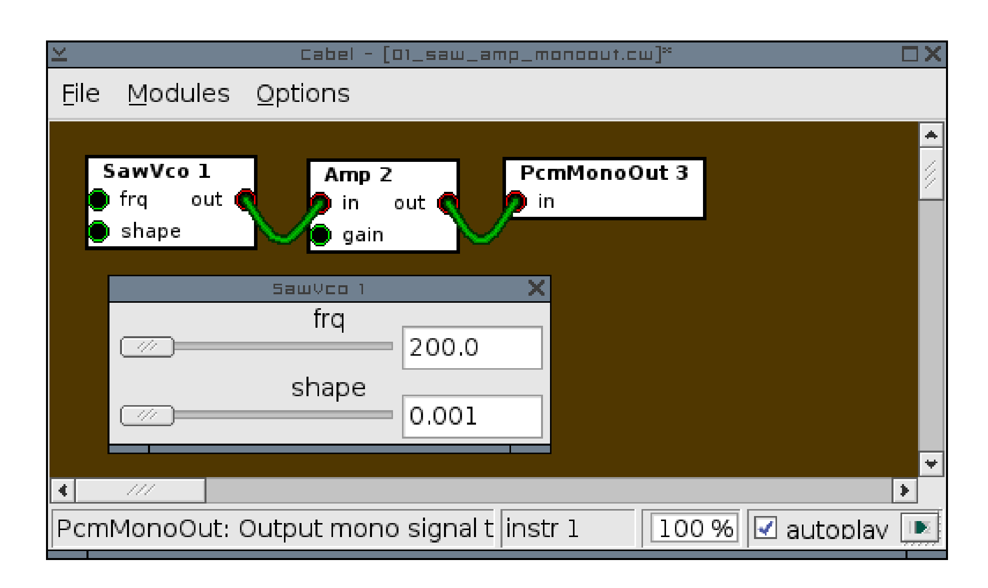


Abbildung 6.1: Das erste Sägezahn Oszillator Instrument

Bevor Sie das Instrument starten überprüfen Sie noch die Einstellungen im Menü *Options*→*Preferences*. Achten Sie darauf, daß *Channels* auf 1 gesetzt ist, da Sie nur ein Mono Signal an die Soundkarte schicken. Falls Sie wollen, daß Ihr Instrument automatisch spielt, setzen Sie Score auf *i1 0 6000*, was so viel wie "spiele Instrument 1 von Anfang an 6000 Sekunden lange" bedeutet. Alternativ können Sie Score auch auf *f0 6000* setzen, was Csound im Hintergrund 6000 Sekunden laufen läßt ohne ein Instrument zu starten. Um ihr Instrument zu triggern reicht ein mit einem MIDI Keyboard an Csound gesendeter MIDI Note On Befehl, also ein einfacher Tastendruck (siehe Kapitel 3.3.1 auf S. 17) aus. Mit welchen *Csound Parametern* Sie Ihre MIDI Geräte mit Csound verbinden können erfahren Sie im Csound Benutzerhandbuch[2].

[1] Verzerrung auf Grund zu hoher Lautstärke
[2] `http://www.csounds.com/manual`

Wenn Sie nun Ihr Instrument mit dem Play Knopf in der Statusleiste starten und falls nötig zusätzlich triggern, sollten Sie einen konstanten obertonreichen Ton hören.

6.1.2 MIDIfizierung des Oszillators

Nun wollen wir die Tonhöhe des Instruments mit unserem MIDI Keyboard steuern. Dazu benötigen Sie ein *MidiNoteIn* Modul aus dem *control* Modulverzeichnis. Dessen *frq* Ausgang verbinden Sie mit dem *frq* Eingang des SawVco.

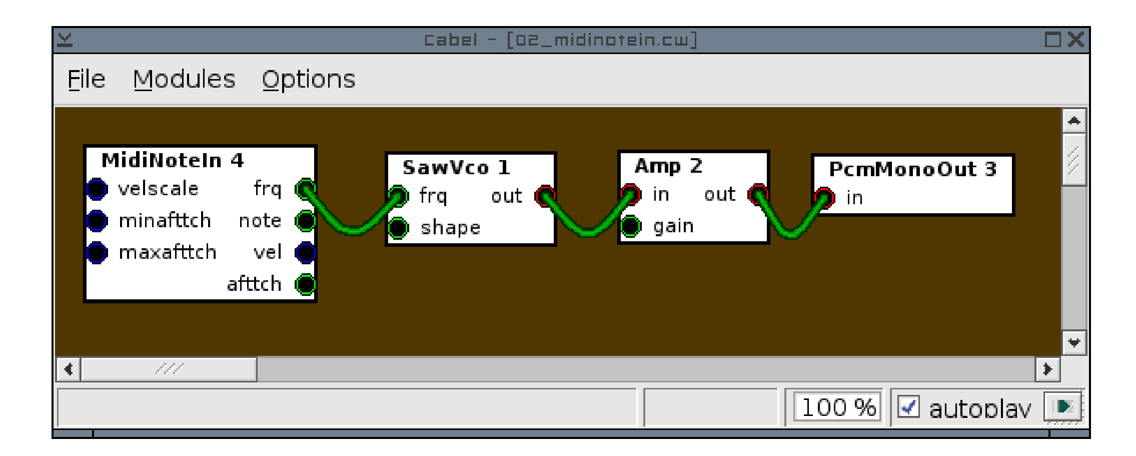


Abbildung 6.2: Steuerung der Tonhöhe mit einem MIDI Keyboard

Stellen Sie sicher, daß das Instrument nicht automatisch getriggert wird, indem Sie den *Score* Parameter im *Preferences* Dialog wie weiter oben beschrieben auf *f0 6000* setzen.

Nun können Sie das Instrument starten und mit Ihrem MIDI Keyboard spielen.

6.1.3 Lautstärkevariation durch eine Hüllkurve

Damit der Ton bei einem Tastendruck nicht abrupt einsetzt und beim Loslassen ebenso abrupt endet, modulieren wir die Lautstärke unseres Instruments mit einer Hüllkurve.

Fügen Sie das Modul *AdsrLinMidi* aus *modulators* zu Ihrer Cabel Arbeitsfläche hinzu und verbinden Sie es wie in Abbildung 6.3 zu sehen.

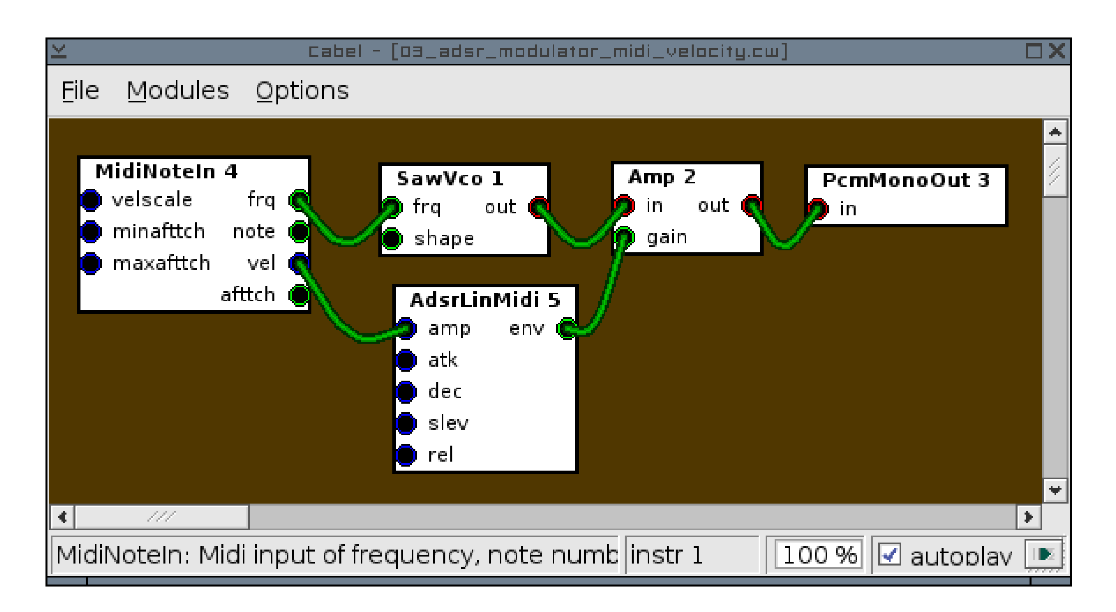


Abbildung 6.3: Modulation mit einem Hüllkurvengenerator

Die maximale Lautstärke unseres Instruments, also den *amp* Parameter des AdsrLinMidi Moduls, erhalten Sie über die Anschlagstärke des MIDI Keyboards, indem Sie den *vel* Ausgang von MidiNoteIn mit dem *amp* Eingang von AdsrLinMidi verbinden und den *velscale* Parameter von MidiNoteIn auf etwa 20000 setzen.

6.1.4 Subtraktive Klangsynthese

Um aus diesem obertonreichen Instrument einen subtraktiven Synthesizer, also einen Klangerzeuger, der unerwünschte Frequenzen aus einem Klang subtrahiert, zu erzeugen, benötigen Sie noch ein Filter, das ungewünschte Frequenzen herausfiltert. Dazu verwenden Sie das Moog Tiefpass Filter, mit dem die Frequenzen, die überhalb der eingestellten *Cutoff Frequenz* liegen, entfernt werden.

Damit der erzeugte Klang nicht zu statisch klingt, wird die *Cutoff Frequenz* des Filters mit einer Hüllkurve moduliert. In Abbildung 6.4 sehen Sie das fertige Instrument.

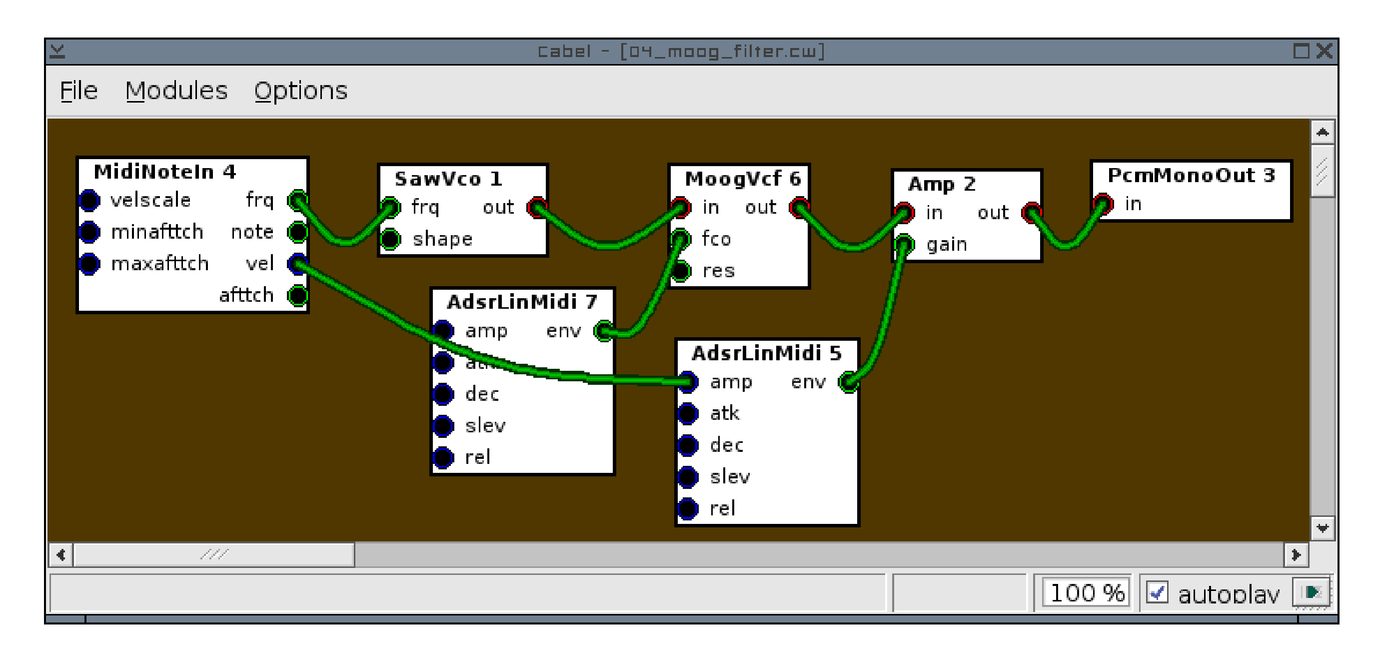


Abbildung 6.4: Einfacher subtraktiver Synthesizer

6.1.5 Erweiterung unseres subtraktiven Synthesizers

In den Beispieldateien *05_suboscillator_mixer2.cw* und *06_detuned_saw_mixer4_analogecho.cw* im *examples* Unterordner Ihrer Cabel Installation können Sie mögliche Erweiterungen Ihres Synthesizers studieren.

Im ersten Schritt wird dem Sägezahn Signal ein um eine Oktave tiefer gestimmter Rechteckoszillator, ein sogenannter Suboszillator, beigemischt, um die Bassfrequenzen unseres Instruments zu verstärken.

Im nächsten Schritt wird zusätzlich noch ein leicht verstimmter Sägezahnoszillator ergänzt, damit der Gesamtklang weniger statisch ist. Um den Klang noch weiter zu verfeinern wird zum Schluß noch alles durch einen Hall Effekt geschickt.

6.2 Verwendung des Sequencers

In diesem kurzen Beispiel wird gezeigt wie das Sequencer Modul verwendet wird. Ausgangspunkt ist dabei der minimale Klangerzeuger aus Kapitel 6.1.1 in der Datei *examples/01_saw_amp_monoout.cw* Ihrer Cabel Installation.

Zunächst fügen Sie zu diesem Instrument ein *Sequencer* Modul aus dem Modulmenü Unterverzeichnis *sequencer* hinzu. Damit dieses Modul in regelmäßigen Zeitabschnitten seine Werte ausgibt, benötigen Sie noch ein Clock Signal. Dazu eignet sich das *PulseLfo* Modul im *modulators* Unterverzeichnis.

Verbinden Sie die Module wie in Abbildung 6.5 zu sehen.

Abbildung 6.5: Verwendung des Sequencer Moduls

Den *Score* Parameter in den Csound Options sollten Sie wie in Kapitel 6.1.1 beschrieben auf *i1 0 6000* setzen, damit das Instrument automatisch getriggert wird. Wenn Sie nun das Instrument starten, hören Sie eine schnell aufsteigende Tonfolge.

Da es schwierig ist eine Notenfolge aus Frequenzen zu konstruieren, können Sie zwischen dem *Sequencer* und dem *SawVco* Modul ein zusätzliches *Midi2Frq* Modul (*maths* Unterverzeichnis) verwenden, damit Sie die Sequenz als Folge von MIDI Notennummern[3] programmieren können und diese dann in die entsprechenden Frequenzen umgerechnet werden.

Die Sequenz ändern Sie, indem Sie die Parameter *val1* bis *val8* im *Sequencer* Modul z.B. auf 40, 52, 40, 55, 40, 57, 40, 52 (E, e, E, g, E, a, E, g) setzen. Die Geschwindigkeit, in der die Sequenz abgespielt wird, stellen Sie über den *frq* Parameter des *PulseLfo* Moduls ein. Um fünf Töne pro Sekunde spielen zu lassen ändern Sie die Frequenz des *PulseLfo* auf 5.

Ein fortgeschrittenes Beispiel für die Verwendung von *Sequencer* Modulen können Sie in der Datei *examples/sequencer_notequantizer_sampleandhold.cw* studieren.

[3]Tabelle zur Umrechnung von MIDI Notennummern: `http://www.zem-college.de/midi/mc_taben.htm`

6.3 Frequenzmodulation

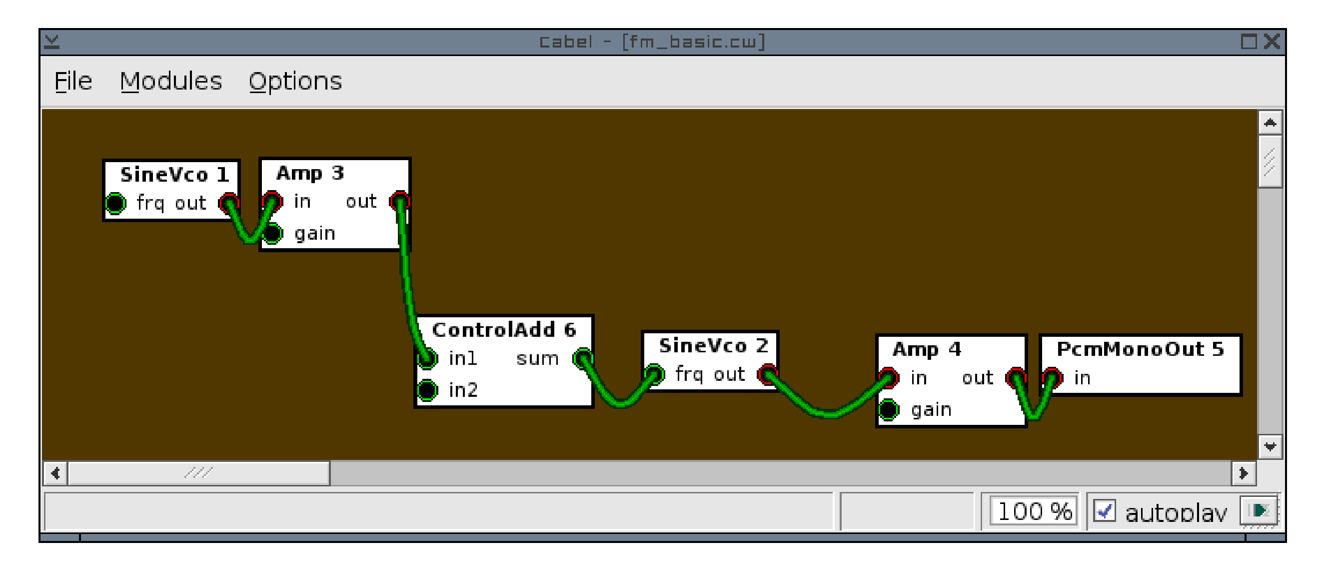


Abbildung 6.6: Frequenzmodulation mit Cabel

Abbildung 6.6 zeigt die einfachste Möglichkeit der Frequenzmodulation: Das verstärkte Signal von *SineVco 1* moduliert die Frequenz von *SineVco2*.

Das Instrument finden Sie unter *examples/fm_basic.cw*. Wenn Sie es starten, hören Sie einen 300 Hz Ton mit einem ziemlich extremen Vibrato. Solange die Frequenz von *SineVco 1* noch unterhalb des hörbaren Frequenzbereichs von 20 Hz liegt, nehmen wir die Frequenzmodulation als Vibrato wahr. Wenn Sie nun mit dem *frq* Parameter von *SineVco 1* und dem *gain* Parameter von *Amp 3* spielen, stellen Sie fest, daß die zwei Sinusoszillatoren in der Lage sind komplexe Frequenzspektren zu erzeugen.

Der Cabel Patch *examples/fm_example.cw*, zu sehen in Abbildung 6.7, ist ein Beispiel für ein FM Instrument. Dabei steuert *SineVco 2* die Frequenz von *SineVco 5*. Die Stärke der Modulation, also die Verstärkung des Signals von *SineVco 2*, wird durch den Hüllkurvengenerator *AdsrLinMidi 20* geregelt, sodaß sich das erzeugte Frequenzspektrum über die Zeit ändert.

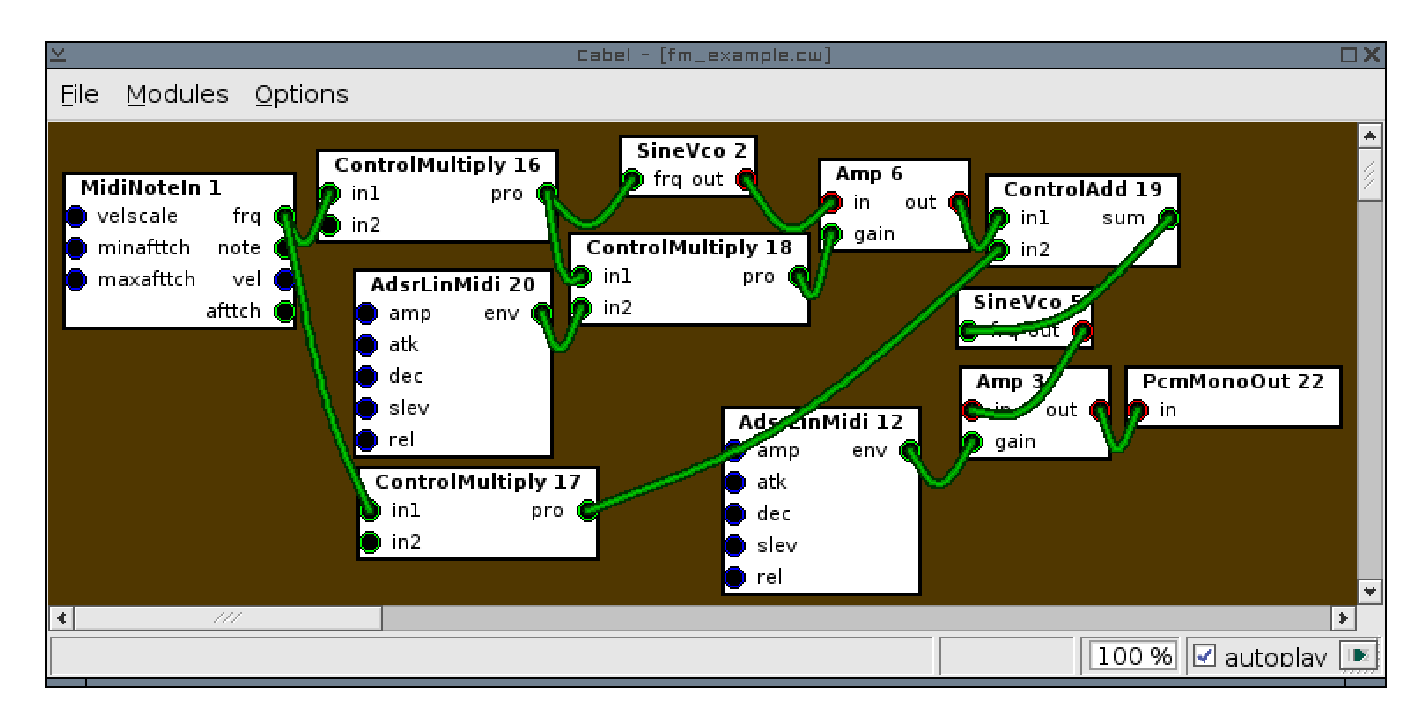


Abbildung 6.7: FM Instrument

6.4 Verarbeitung externer Audio Signale

Sie können Cabel auch als Effektgerät verwenden und externe Audio Signale weiterverarbeiten. Beachten Sie dazu, daß Sie wie in Kapitel 3.3.1 auf S. 18 beschrieben Csound mit den richtigen Parametern starten, um Audio Signale in Echtzeit von der Soundkarte lesen zu können. Vergessen Sie auch nicht den *Score* Parameter wie in Kapitel 3.3.1 auf S. 17 beschrieben so einzustellen, daß Ihr Instrument automatisch getriggert wird.

In Abbildung 6.8 sehen Sie ein Beispiel, in dem ein externes Stereo Audio Signal mit einer Rechteckswelle multipliziert wird, also mit einem *SquareVco* ringmoduliert wird. Mit dem MIDI Controller 1 (*MidiCtrlIn 11*) können Sie den Anteil des ringmodulierten Signals im Verhältnis zum Originalsignal steuern. MIDI Controller 2 (*MidiCtrlIn 7*) steuert die Frequenz der Rechteckwelle, und MIDI Controller 3 (*MidiCtrlIn 8*) die Lautstärke von *SquareVco* und damit die Intensität der Ringmodulation.

Das Beispiel finden Sie in der Datei *examples/external_square_ringmod.cw* in Ihrer Cabel Installation.

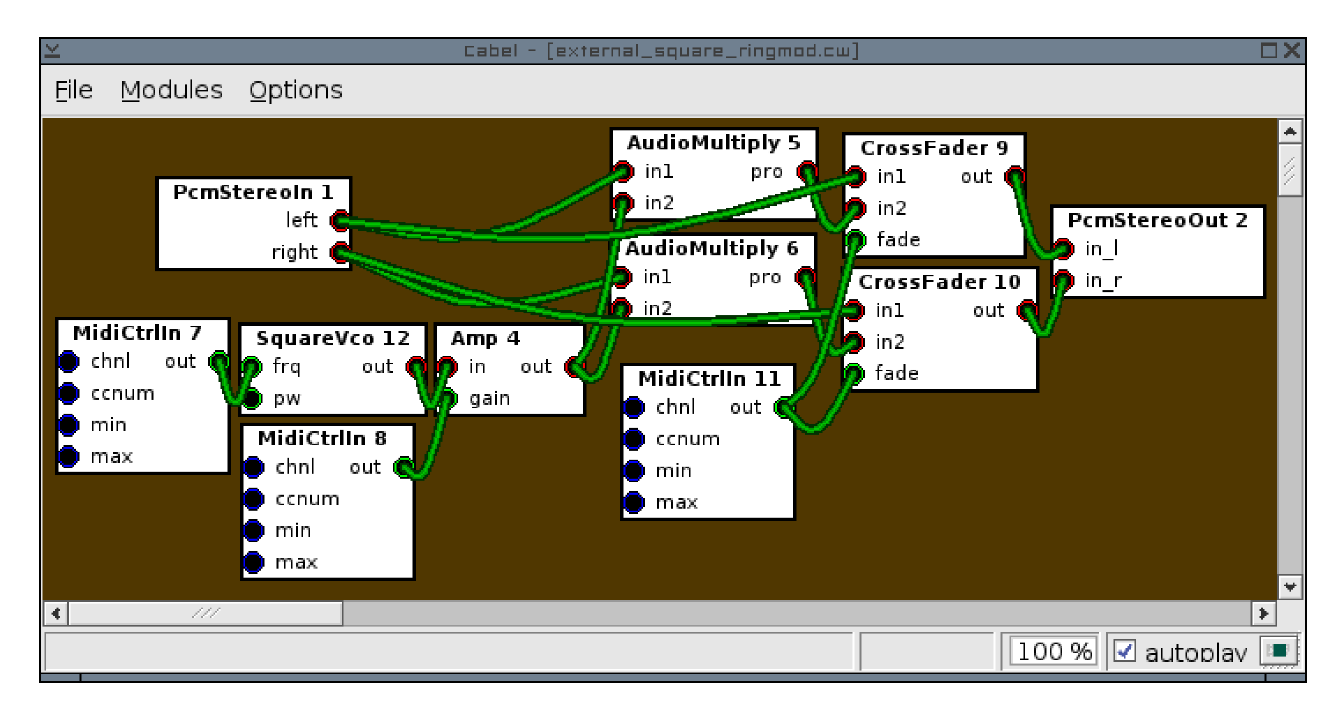


Abbildung 6.8: Ringmodulation eines externen Audio Signals mit einem *SquareVco*

Kapitel 7

Schreiben eigener Module

Wie bereits in der Einleitung (siehe Kapitel 1.1.2 auf S. 7) erwähnt, wird jedes in Cabel verwendbare Modul in einer eigenen XML[1] Datei definiert.

In folgendem Kapitel wird kurz auf den Aufbau dieser Cabel XML Module eingegangen.

7.1 Kurze Einführung zu XML

Eine XML Datei ist ein hierarchisch in Form einer Baumstruktur organisiertes Dokument. Die Elemente des Dokumentenbaums werden als *XML-Nodes*[2] bezeichnet. Ein XML Node besteht entweder aus einem Start-Tag und einem zugehörigen Ende-Tag:

```
<TagName attributName="AttributWert">TagWert</TagName>
```

oder aus einem leeren Element-Tag:

```
<TagName attributName="Attribut Wert"/>
```

Auf diese Weise kann man Dokumente einfach gliedern und deren Inhalt in eine Struktur bringen, die sowohl für den Menschen auch als auch für einen Computer lesbar und verwertbar ist.

Der konkrete Aufbau von XML-Dateien kann durch sogenannte XML-Schema Dateien festgelegt werden. Erfüllt die XML Datei die im Schema festgelegte Syntax, so spricht man von einem *gültigen* oder auch *validen* XML-Dokument.

[1] eXtensible Markup Language, ein Standard zur Erstellung von maschinen- und menschenlesbaren Dokumenten.

[2] engl. Knoten

Diese Schema Datei wird in der XML-Datei nach der Definition des in der Datei verwendeten Encodings[3] (Zeile 1 in Listing 7.1) im Dokumentroot Knoten[4] (in Listing 7.1 in Zeile 2 der `<modul>` Knoten) angegeben.

Auf diese Weise können Programme, die XML Dateien verarbeiten, einfach überprüfen, ob die übergebenen XML Dateien gültig sind. Diesen Prozess nennt man auch *Validieren von XML Dateien*.

Die meisten aktuellen XML Editoren[5] verstehen XML-Schema und validieren XML Dokumente bereits beim Erstellen. Dadurch werden die für den Menschen relativ schwer lesbaren XML Schemas zu so etwas wie einer Vorlage und erleichtern das Anlegen eigener XML Dateien.

Weitere Informationen rund um die *eXtendable Markup Language* finden Sie unter

- `http://de.selfhtml.org/xml/intro.htm`, oder
- `http://de.wikipedia.org/wiki/XML`.

7.2 Aufbau der Cabel Modul XML Dateien

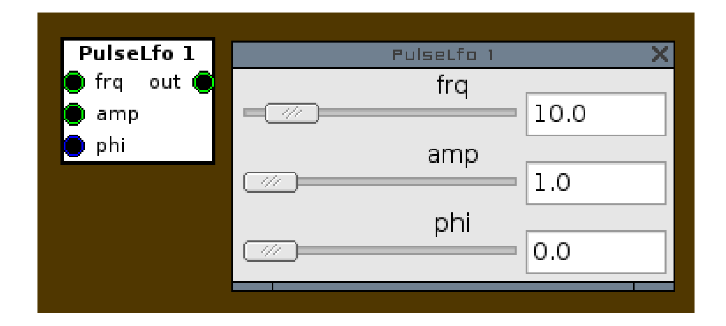


Abbildung 7.1: Das in Listing 7.1 beschriebene PulseLFO Modul mit zugehörigem Wertefenster in der Cabel Arbeitsfläche.

Listing 7.1: Die XML Datei für das Cabel Modul *PulseLFO*

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <modul xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3        xsi:noNamespaceSchemaLocation="modul.xsd"
4        name="PulseLfo"
5        description="Pulse␣low␣frequency␣oscillator␣(clock␣signal)">
6   <input>
```

[3]Encodings legen fest, welcher Zeichenvorrat in der Datei verwendet wird

[4]Erster, alle weiteren XML Knoten umschliessender Knoten

[5]Unter Windows z.B. *XMLSpy* von Altova (`http://www.altova.com/de/`)

```
7     <vardef name="frq" csType="k" value="10"
8             min="0" max="100" description="Frequency␣of␣LFO"/>
9     <vardef name="amp" csType="k" value="1"
10            min="0" description="Amplitude␣of␣LFO"/>
11    <vardef name="phi" csType="i" value="0" min="-0.001" max="1"
12            digits="3" description="Initial␣phase␣of␣LFO␣(negativ␣\\
13 to␣skip␣initialisation)"/>
14   </input>
15   <output>
16     <vardef name="out" csType="k" description="Output␣of␣LFO"/>
17   </output>
18   <global>
19     <def description="Pulse␣table">
20       gipulselfo   ftgen  0, 0, 128, -7, 1, 4, 1, 0, 0, 124, 0
21     </def>
22   </global>
23   <opcode>/* Pulse low frequency oscillator (clock signal) */
24 opcode PulseLfo, k, kki
25   kfrq, kamp, iphi xin
26
27   kfrq     limit     kfrq, 0, 100
28   kamp     limit     kamp, 0, 50000
29   iphi     limit     iphi, -1, 1
30   kout     oscil     kamp, kfrq, gipulselfo, iphi
31
32   xout kout
33 endop
34   </opcode>
35 </modul>
```

7.2.1 Der Modul Knoten

Der Dokumentroot Knoten, der alle restlichen Elemente des Cabel XML Moduls umschließt, hat den Namen `modul` (in Listing 7.1 Zeile 2-5 und Zeile 35).

In diesem Element wird in entsprechenden Attributen das Schema (`noNamespaceSchemaLocation`), der Name des Moduls (`name`) und eine kurze von Cabel dann in der Statusleiste[6] angezeigte Beschreibung des Moduls (`description`) angegeben.

Dabei ist es wichtig, daß der Wert des *name* Attributs sowohl dem Dateinamen des Moduls als auch dem Namen des User-defined Opcodes, der im *opcode* Knoten (Kapitel 7.2.5) definiert wird, entspricht (siehe Zeile 24 in Listing 7.1).

[6] Siehe Abbildung 4.6 auf Seite 30.

7.2.2 Eingabe Variablen

Die Eingabe Variablen des Moduls werden im Knoten `input` (in Listing 7.1 Zeile 6-14) definiert. Dieser enthält für jeden Eingabeparameter des Moduls einen Knoten `vardef`, der folgende Attribute besitzt:

- `name`*[7] Anzeige Name der Eingabe Variable auf dem Cabel Modul.
- `csType`* Wie in Kapitel 4.2.1.1 beschrieben, der Csound-Typ[8] der Eingabe Variablen; einer der Werte `a` (für Audio Signal), `k` (für Kontroll Signal), oder `i` (für Instrument Signal).
- `description` Beschreibung der Eingabe Variable.
- `value` Standardwert, mit dem die Eingabe Variable initialisiert wird.
- `min` Untere Grenze des gültigen Wertebereichs der Eingabe Variable.
- `max` Obere Grenze des gültigen Wertebereichs der Eingabe Variable.
- `digits` Anzahl der Dezimalstellen, auf die der Wert der Eingabe Variable gerundet werden soll.

Reihenfolge und Typ der einzelnen Eingabeparameter muß den Parametern des User-defined Opcodes im Opcode Knoten (Kapitel 7.2.5) entsprechen.

7.2.3 Ausgabe Variablen

Die Ausgabe Variablen des Moduls werden im Knoten `output` (in Listing 7.1 Zeile 15-17) definiert. Dieser enthält für jeden Ausgabeparameter des Moduls einen Knoten `vardef`, der folgende Attribute besitzt:

- `name`* Anzeige Name der Ausgabe Variable auf dem Cabel Modul.
- `csType`* Wie in Kapitel 4.2.1.1 beschrieben, der Csound-Typ[9] der Ausgangs Variablen; einer der Werte `a` (für Audio Signal), `k` (für Kontroll Signal), oder `i` (für Instrument Signal).
- `description` Beschreibung der Eingabe Variable.

Reihenfolge und Typ der einzelnen Ausgabeparameter muß den Parametern des User-defined Opcodes im Opcode Knoten (Kapitel 7.2.5) entsprechen.

[7]Attribute, die mit einem * versehen sind, sind verpflichtende Attribute, die für ihren Knoten angegeben werden müssen, damit das XML Modul valide ist.

[8]Rate, in der das Eingangssignal verarbeitet werden soll

[9]Rate, in der das Ausgangssignal verarbeitet werden soll

7.2.4 Globale Variablen

In Cabel Modulen können auch so genannte *globale Variablen* definiert werden. Diese werden in dem Cabel XML Knoten `global` (in Listing 7.1 Zeile 18-22) angegeben.

Globale Variablen dienen vor Allem dazu, Funktionstabellen[10] einmalig global zu definieren. Dies spart Speicherplatz, da sie nicht von jeder Modul Instanz auf der Arbeitsfläche neu initialisiert werden müssen, sondern *global* für das ganze Instrument gültig und ansprechbar sind.

Der Knoten `global` enthält für jede globale Variable einen Konten `def`.

Diese `def` Knoten sind Text-Nodes, deren einziges Attribut das optionale Attribut `description` ist. Es dient nur der Dokumentation des Moduls und soll die globale Variable kurz beschreiben. Der eigentliche Inhalt des `def` Knotens ist ein Csound Opcode als Text zwischen dem Start- und Ende-Tag des `def` Elements.

Als Beispiel für die Definition einer *globalen* Variable siehe Zeile 18 in Listing 7.1.
Für nähere Informationen zum darin verwendeten Csound Opcode `ftgen` siehe `http://www.csounds.com/manual/html/ftgen.html`.

7.2.5 Opcode Knoten

Schließlich folgt der Knoten `opcode` (Listing 7.1 ab Zeile 23), in dem die eigentlichen Csound Anweisungen als reiner Text definieren, was das Modul zu machen hat.

Die hier anzugebende Anweisung ist die Definition eines User-defined Opcodes[11]. Die Syntax dieses Codes hier zu erklären würde zu weit führen; dafür sei auf die folgenden Seiten im Internet verwiesen:

- Opcode Definition: `http://www.csounds.com/manual/html/opcode.html` und
- User-defined Opcode Bibliothek: `http://www.csounds.com/udo`.

7.3 Einbinden der XML Modul Dateien in Cabel

Um ein wie in den letzten Abschnitten beschriebenes eigenes Cabel XML Modul auch in Cabel verwenden zu können, muß eine entsprechende XML Datei im *Modul Verzeichnis*[12], oder einem Unterverzeichnis davon angelegt werden. Zum Erstellen wird ein XML Editor empfohlen, der, wie in Kapitel 7.1 auf Seite 56 beschrieben, das selbst geschriebene Modul gegen das Schema `modul.xsd` (zu finden im *Modul Verzeichnis*) validieren kann.

[10] Für nähere Information siehe `http://www.csounds.com/manual/html/f.html`

[11] engl. Benutzerdefinierter Opcode

[12] siehe Kapitel 3.3.3 auf Seite 22

Haben sie ein valides Cabel XML Modul dort gespeichert, erscheint es in der Cabel Modul Liste, nachdem Sie unter *Options→Refresh Module List* die Liste der Cabel Module aktualisiert haben (Beschrieben in Kapitel 3.2 auf Seite 16).

7.3.1 Skinning von Cabel Modulen

Sie können für jedes Cabel Modul ein Hintergrundbild festlegen und somit das Erscheinungsbild Ihrer Module gestalten.

Dazu legen Sie eine JPEG-Bilddatei[13] mit demselben Namen des Moduls, das Sie skinnen möchten, in dessen Modul Verzeichnis ab, also z.B. *modulators/PulseLfo.jpg* für *modulators/PulseLfo.xml*. Das Bild wird zentriert unter die Darstellung des Moduls gesetzt und, falls das Bild größer als das Modul ist, wird die Modulgröße an die des Bildes angepaßt.

[13]Joint Photographic Experts Group; Bilddatei mit der Endung *JPG*.

Kapitel 8

Automatisierung von Cabel

Cabel kann mit Hilfe von Python[1] automatisiert werden. Öffnen Sie über das Menü mit *Options→Show Bottom Pane* die *Python Shell*.

Über die Variable *w* haben sie Zugriff auf die Funktionen der Klasse *model.workspace* (siehe Anhang A.7, S. 81), die die Funktionalität von Cabel implementiert. Weitere Information zu den Funktionen von *model.workspace* entnehmen Sie bitte der Cabel API[2] in Anhang A.

8.1 Generierung eines Instruments

Wenn Sie Listing 8.1 in die Python Shell eingeben, sollten Sie nach dem Triggern des Instruments einen Sinuston mit 440 Hz hören.

Listing 8.1: Sinuston

```
1  sine = w.addXmlModule("sound_sources/SineVco")
2  amp = w.addXmlModule("amps_mixers/Amp")
3  out = w.addXmlModule("input_output/PcmMonoOut")
4  w.connect(sine.outVars[0], amp.inVars[0])
5  w.connect(amp.outVars[0], out.inVars[0])
6  w.setValue(sine.inVars[0], 440)
7  w.setValue(amp.inVars[1], 20000)
8  w.play()
```

Mit *w.stop()* können Sie das Instrument wieder stoppen.

In Zeile 1 bis 3 werden die einzelnen Module erzeugt. In Zeile 4 wird der erste Ausgang des *SineVco* Moduls (Index 0) mit dem ersten Eingang des *Amp* Moduls (Index 0) ver-

[1] Falls Sie noch nie mit Python gearbeitet haben, empfehlen wir das Buch "Dive into Python" [Pil04]
[2] Application Programming Interface

bunden. In Zeile 5 entsprechend der erste Ausgang von *Amp* mit dem ersten Ausgang von *PcmMonoOut*. In den darauffolgenden Zeilen werden den nicht verbundenen Eingängen Werte zugewiesen und das Instrument gestartet.

8.2 Vereinfachung sich wiederholender Arbeitsschritte

Falls Sie eine Folge von Arbeitsschritten für Ihren Instrumententwurf mehrmals durchführen müssen, können Sie diese ebenfalls automatisieren.

Dazu ein kleines Beispiel: Um drei durch *TriangleLfo* frequenzmodulierte Moog Filter zu erzeugen, verwenden Sie den Code in Listing 8.2.

Listing 8.2: Drei durch *TriangleLfo* modulierte Moog Filter

```
1 for i in xrange(3):
2     lfo = w.addXmlModule("modulators/TriangleLfo")
3     moog = w.addXmlModule("filters/MoogVcf")
4     w.connect(lfo.outVars[0], moog.inVars[1])
```

8.3 Einbindung externer Python Skripte

Zuletzt wollen wir Cabel durch ein externes Skript automatisieren. Erzeugen Sie in ihrem Cabel Verzeichnis eine Datei *script.py* mit dem Inhalt von Listing 8.3.

Listing 8.3: Ein externes Python Skript

```
1 def sine(w, frq):
2     sine = w.addXmlModule("sound_sources/SineVco")
3     amp = w.addXmlModule("amps_mixers/Amp")
4     out = w.addXmlModule("input_output/PcmMonoOut")
5     w.connect(sine.outVars[0], amp.inVars[0])
6     w.connect(amp.outVars[0], out.inVars[0])
7     w.setValue(sine.inVars[0], frq)
8     w.setValue(amp.inVars[1], 20000)
```

Diese Datei definiert eine Funktion, die ein Instrument ähnlich Listing 8.1 erzeugt. Geben Sie in der *Python Shell* Listing 8.4 ein um dieses Skript zu importieren und ein Sinus Instrument mit einer Frequenz von 300 Hz zu erzeugen.

Listing 8.4: Ausführen eines externen Skriptes

```
1 import script
2 script.sine(w, 300)
```

Anhand dieser Beispiele und mit Hilfe der Cabel API in Anhang A sollten Sie in der Lage sein, Cabel zu automatisieren.

Anhang A

Cabel API

A.1 Module model.connection

A.1.1 Class Connection

__builtin__.object ─┐
Connection

Connection between two modules.

A Connection is defined by its start Var and its end Var.

A.1.1.1 Methods

__init__(*self*, *fromVar*, *toVar*)

Standard constructor.

Parameters

`fromVar`: Start Var of this connection.
(type=model.var.OutVar)

`toVar`: End Var of this connection.
(type=model.var.InVar)

Overrides: __builtin__.object.__init__

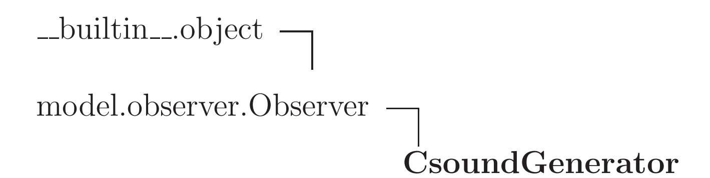
Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.1.1.2 Instance Variables

Name	Description
fromVar	Start Var of this connection. *(type=model.var.OutVar)*
toVar	End Var of this connection. *(type=model.var.InVar)*

A.2 Module model.csound

A.2.1 Class CsoundGenerator



CsoundGenerator.

Generates csound code from actual state of workspace.

A.2.1.1 Methods

__init__(*self*, *workspace*)

Standard constructor.

Parameters

`workspace`: Workspace on which the CsoundGenerator works.
(type=model.workspace.Workspace)

Overrides: model.observer.Observer.__init__

exportToCsd(*self*, *filePath*)

Export options, orchestra and score into CSD file.

Parameters

`filePath`: Path to CSD file (relative or absolute).
(type=str)

generate(*self*)

Generates the Csound Code out of the in the workspace bundled objects.

Return Value

CSound code.
(type=string)

play(*self*)

Starts CSD generation and csound in a seperate process. If csound's already playing stops it first.

Return Value

True if the csound process started successfully.
(type=boolean)

stop(*self*)

Stops running Csound process.

Return Value

True if there was a running csound process to be stopped, else False.
(type=boolean)

update(*self*, *observable*, *arg*)

This method is called whenever the observed object is changed. An application calls an observable object's notifyObservers method to have all the object's observers notified of the change.

Parameters

`arg`: An argument passed to the notifyObservers method.
(type=object)

Overrides: model.observer.Observer.update

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.2.1.2 Instance Variables

Name	Description
config	Defines and encapsulates user specified csound relevant variables. (see config.xml) *(type=tools.config.Config)*
csoundVars	Defines and encapsulates user specified csound relevant variables. (see config.xml) *(type=tools.config.Csound)*
workspace	Link to workspace object. *(type=model.workspace.Workspace)*

A.3 Module model.instrument

A.3.1 Class Instrument

```
__builtin__.object ─┐
                    Instrument
```

Instrument.

An Instrument contains a list of included modules and their connections.

A.3.1.1 Methods

__init__(*self*)
Standard constructor. Overrides: __builtin__.object.__init__

printConnections(*self*)
Print connections and their indices to standard output.

printModules(*self*)
Print contained modules and their indices to standard output.

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.3.1.2 Instance Variables

Name	Description
connections	Connections between intrument modules. *(type=list)*
modules	Modules contained in intrument. *(type=list)*

A.4 Module model.module

A.4.1 Class Module

__builtin__.object ─┐
Module

Module.

A Module contains its user-defined opcode text and lists of its In- and OutVars.

A.4.1.1 Methods

__init__(*self*, *name*, *description*='', *opcode*='', *_id*=0)

Standard constructor.

Parameters

`name:` Name of module/user-defined opcode
(type=string)

`description:` describes module.
(type=string)

`opcode:` Text of user defined opcode for name
(type=string)

`_id:` Unique id for module.
(type=integer)

Overrides: __builtin__.object.__init__

addGlobal(*self*, *globus*, *description*)

add globus (global is a fukin' keyword) to the list of the module's global-vars.
nice to save tables in, so that the csoundGenerator can extract them and doesn't have to generate one for every instance of the module.

Parameters

`globus:` new global-var for the module as crude csound-code
(type=str)

`description:` description for the global-var
(type=str)

addInVar(*self*, *inVar*)

add inVar to the list of the module's input-vars.

Parameters

`inVar`: new input to the module
(type=model.var.inVar)

addOutVar(*self*, *outVar*)

add outVar to the list of the module's output-vars.

Parameters

`outVar`: new output of the module
(type=model.var.outVar)

getGlobalAsCsoundCode(*self*, *index*)

Returns the GlobalVar definition as a crude csound-code string. If there is a description stored it is appended as a comment.

Parameters

`index`: The index of the global.
(type=int)

Return Value

The global var definition as crude csound-code string with or without comment.
(type=str)

printInputs(*self*)

Print input names, indices, their values and value ranges to standard output.

printOutputs(*self*)

Print output names and indices to standard output.

setInVarValues(*self*, *values*)

Set inVar values.

Parameters

`values`: List of tuples each describing a var by its id and value.
(type=list)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.4.1.2 Instance Variables

Name	Description
description	describes module. *(type=string)*
fullName	Name of module including relative path *(type=string)*
globals	List of Strings for global statements. *(type=list)*
id	Unique id for module. *(type=integer)*
instrument	Instrument containing this module. *(type=model.instrument.Instrument)*
inVars	List of input variables. *(type=list)*
name	Name of module/user-defined opcode *(type=string)*
opcode	Text of user defined opcode for name *(type=string)*
outVars	List of output variables. *(type=list)*

A.5 Module model.observer

A.5.1 Class Observable

__builtin__.object
└ **Observable**

Known Subclasses: Workspace

Observable.

This class represents an observable object, or 'data' in the model-view paradigm. It can be subclassed to represent an object that the application wants to have observed. An observable object can have one or more observers. After an observable instance changes, an application calling the Observable's notifyObservers method causes all of its observers to be notified of the change by a call to their update method.

A.5.1.1 Methods

__init__(*self*)

Standard constructor.

Overrides: __builtin__.object.__init__

---

addObserver(*self, observer*)

Adds a new observer to the list of observers.

Parameters

`observer`: New observer to add.
(type=Observer)

---

clearChanged(*self*)

Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change. This method is called automatically by the notifyObservers methods.

---

countObservers(*self*)

Returns the number of observers of this object.
@rtype : int

Return Value

Number of observers.

deleteObservers(*self*)

Clears the observer list so that this object no longer has any observers.

hasChanged(*self*)

Tests if this object has changed.
@rtype : bool

Return Value

If this object has changed.

notifyObservers(*self*, *arg*=`None`)

If this object has changed, as indicated by the hasChanged method, then notify all of its observers and then call the clearChanged method to indicate that this object has no longer changed. Each observer has its update method called with two arguments: this observable object and the arg argument.

Parameters

`arg`: Data representing the change in Observable.
(type=object)

removeObserver(*self*, *observer*)

Remove observer from list of observers.

Parameters

`observer`: Observer to remove.
(type=Observer)

resumeObservation(*self*)

setChanged(*self*)

Indicated that this object has changed.

suspendObservation(*self*)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.5.2 Class Observer

__builtin__.object
└ **Observer**

Known Subclasses: CabelController, CabelFrame, CabelIOTextCtrl, CabelStatusBar, CabelValueFrame, CsoundGenerator

Observer.

A class can inherit from Observer interface when it wants to be informed of changes in observable objects.

A.5.2.1 Methods

__init__(*self*)

Standard constructor.

Overrides: __builtin__.object.__init__

update(*self*, *observable*, *arg*)

This method is called whenever the observed object is changed. An application calls an observable object's notifyObservers method to have all the object's observers notified of the change.

Parameters

`observable`: The observable object.
(type=Observable)

`arg`: An argument passed to the notifyObservers method.
(type=object)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.6 Module model.var

A.6.1 Class InVar

```
__builtin__.object ─┐
                    │
        model.var.Var ─┐
                       │
                     InVar
```

Module input variable.

Special Var for input variables, which can set its actual value in the range min to max if not connected.

A.6.1.1 Methods

__init__(*self*, *module*, *name*, *_type*, *description*=`''`, *_min*=`-32768`, *_max*=`32767`, *value*=`0`, *digits*=`3`)

Standard constructor.

Parameters

- `module`: Module to which this variable belongs.
  (type=model.module.Module)
- `name`: Name of variable.
  (type=string)
- `_type`: Type of variable.
  (type=string)
- `_min`: Minimal allowed value for value.
  (type=float)
- `_max`: Maximal allowed value for value.
  (type=float)
- `value`: Initial value of variable.
  (type=float)
- `digits`: Number of digits (or presicion) for the value.
  (type=int)

Overrides: model.var.Var.__init__

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.6.1.2 Properties

Name	Description
Value	

A.6.1.3 Instance Variables

Name	Description
connection	Link to connection to this var. *(type=model.connection.Connection)*
digits	Number of digits (or presicion) for the value. *(type=int)*
max	Maximal allowed value for value. *(type=float)*
min	Minimal allowed value for value. *(type=float)*
value	Initial value of variable. *(type=float)*
Inherited from Var: module *(p. 78)*, name *(p. 78)*, type *(p. 78)*	

A.6.2 Class OutVar

```
__builtin__.object ─┐
                    │
         model.var.Var ─┐
                        │
                     OutVar
```

Module output variable.

At the moment this is just a wrapper for Var with special name.

A.6.2.1 Methods

__init__(*self*, *module*, *name*, *_type*, *description*=`''`)

Standard constructor.

Parameters

- `module`: Module to which this variable belongs.
  (type=model.module.Module)
- `name`: Name of variable.
  (type=string)
- `_type`: Type of variable.
  (type=string)

Overrides: model.var.Var.__init__

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.6.2.2 Instance Variables

Name	Description
description	describes the meaning of the input-var *(type=string)*
Inherited from Var: module *(p. 78)*, name *(p. 78)*, type *(p. 78)*	

A.6.3 Class Var

__builtin__.object
 └ **Var**

Known Subclasses: InVar, OutVar

Module variable.

Var contains its corresponding module, variable name and variable type. This class serves as interface for special variables.

A.6.3.1 Methods

__init__(*self*, *module*, *name*, *_type*, *description=''*)

Standard constructor.

Parameters

`module`: Module to which this variable belongs.
(type=model.module.Module)

`name`: Name of variable.
(type=string)

`_type`: Type of variable.
(type=string)

`description`: describes the meaning of variable.
(type=string)

Overrides: __builtin__.object.__init__

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.6.3.2 Instance Variables

Name	Description
module	Module to which this variable belongs. *(type=Module)*
name	Name of variable. *(type=string)*
type	Type of variable. *(type=string)*

A.6.4 Class VarValueOutOfRangeError

exceptions.Exception
  └ **VarValueOutOfRangeError**

VarValueOutOfRangeError.

Exception if new value for InVar is out of rage.

A.6.4.1 Methods

Inherited from Exception: __init__, __getitem__, __str__

A.7 Module model.workspace

A.7.1 Class ConnectionError

exceptions.Exception ─┐
ConnectionError

ConnectionError.

Exception if connection fails.

A.7.1.1 Methods

Inherited from Exception: __init__, __getitem__, __str__

A.7.2 Class Workspace

__builtin__.object ─┐
model.observer.Observable ─┐
Workspace

Workspace.

Manages list of instruments and connections between modules and module-IDs.

A.7.2.1 Methods

__init__(*self*)

Standard constructor.

Overrides: model.observer.Observable.__init__

addModule(*self, module*)

Add module to workspace and create a new instrument which includes this module. Adds entry in self._modulesDict dictionary.

Parameters
`module`: Module to be added.
(type=model.module.Module)

addXmlModule(*self*, *moduleName*)

Reads an xml-module with an unique id and adds it to the workspace.

Parameters

`moduleName`: the name of the module which should be defined in a moduleName.xml File in the searchPath of the moduleReader ('modules/' by default)
(type=str)

Return Value

Returns added module.

Raises

`model.xmlReader.ModuleNotFoundError` if the given moduleName-module couldn't be found in the searchpath

`model.xmlReader.ModuleDefinitionError` if the given moduleName-xml-module wasn't valid @rtype : model.module.Module

connect(*self*, *fromVar*, *toVar*)

Connect fromVar to toVar, add all modules of toVar-module-instrument to instrument of fromVar-module and sort modules list in the correct order for csound code generation.

Parameters

`fromVar`: Startpoint of connection.
(type=model.var.OutVar)

`toVar`: Endpoint of connection.
(type=model.var.InVar)

Return Value

New connection.

Raises

`ConnectionError` If connection from module to itself. @rtype : model.Connection.connection

connect2(*self*, *fromModuleId*, *fromVarId*, *toModuleId*, *toVarId*)

Same implementation, different call than connect(fromVar, toVar).

Parameters

`fromModuleId`: Module id in the workspace.
(type=int)

`fromVarId`: Output number of the fromModule.
(type=int)

`toModuleId`: Module id in the workspace.
(type=int)

`toVarId`: Input number of the fromModule.
(type=int)

createWorkspace(*self*, *workspaceReader*=`None`)

Create a Workspace from a workspaceReader object or a new, empty one. Informes observers.

Parameters

`workspaceReader`: Reads a complete cabel workspace from .cw files.
(type=model.xmlReader.XmlWorkspaceReader)

disconnect(*self*, *connection*)

Delete connection. Test if we need to create a new instrument for the disconnected modules.

Parameters

`connection`: Connection which should be deleted.
(type=model.connection.Connection)

getModuleById(*self*, *id*)

Returns module with this id.

Parameters

`id`: Id of wanted module. @rtype : model.module.Module
(type=int)

Return Value

Module with this id or None if not found.

getModuleDescription(*self*, *module*)

Return description string of module.

Parameters

`module`: Module whose description we want. @rtype : str
(type=model.module.Module)

Return Value

Description of module.

getValue(*self*, *var*)

Return actual value of var.

Parameters

`var`: Variable whose value we want. @rtype : float
(type=model.var.InVar)

Return Value

Value of var.

getVarDescription(*self*, *var*)

Return description string of variable.

Parameters

`var`: Module whose description we want. @rtype : str
(type=model.module.Module)

Return Value

Description of module.

isPlaying(*self*)

Return wether cabel is in playing state or not.

play(*self*)

Starts csound in separate process.

printInstruments(*self*)

Print instruments with their contained modules and indices to standard output.

removeModule(*self*, *module*)

Remove module and all its ingoing and outgoing connections from workspace. Removes entry in self._modulesDict dictionary.

Parameters

`module`: Module which should be removed.
(type=model.module.Module)

setIoTextCtrl(*self*, *control*)

setValue(*self*, *var*, *value*)

Set var to value.

Parameters

`var`: Variable which we want to change.
(type=model.var.InVar)

`value`: New value for variable.
(type=float)

Raises

`VarValueOutOfRangeError` If new value is out of range.

stop(*self*)

Stops csound process.

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

Inherited from Observable: addObserver, clearChanged, countObservers, deleteObservers, hasChanged, notifyObservers, removeObserver, resumeObservation, setChanged, suspendObservation

A.7.2.2 Instance Variables

Name	Description
instruments	List of intruments. *(type=list)*

A.7.2.3 Class Variables

Name	Description
config	Interface to config categories. **Value:** `<tools.config.Config instance at 0x01879288>` *(type=tools.config.Config)*

A.8 Module model.xmlGenerator

A.8.1 Class XmlGenerator

__builtin__.object
└ **XmlGenerator**

XmlGenerator.

Generates Cabel Xml.

A.8.1.1 Methods

__init__(*self*, *fileLocation*)

Standardconstructor.

Parameters

`fileLocation`: Complete path to the save file.
(type=string)

Overrides: __builtin__.object.__init__

getXml(*self*, *workspace*)

Returns a Xml node with reference-, instance- and connections-nodes.

Parameters

`workspace`: *(type=view.workspace.CabelFrame)*

Return Value

Xml node.
(type=xml.dom.Element)

getXmlAdditionalInfo(*self*, *workspace*)

Returns node with addiotional Information to be saved.

Parameters

`workspace`: *(type=view.workspace.CabelFrame)*

Return Value

Xml node.
(type=xml.dom.Element)

getXmlConnections(*self*, *cons*)

Returns Xml connection nodes

Parameters

`cons`: List of view connections.
(type=list)

Return Value

Xml connections node.
(type=xml.dom.Element)

getXmlModuleInstance(*self*, *mod*)

Returns a Xml module instance node

Parameters

`mod`: View module to create the instance node.
(type=view.module.Module)

Return Value

Xml module instance node.
(type=xml.dom.Element)

getXmlModuleReference(*self*, *mod*)

Returns a xml module reference node.

Parameters

`mod`: Model module to create the reference node.
(type=model.module.Module)

Return Value

Xml module reference node.
(type=xml.dom.Element)

writeWorkspace(*self*, *workspace*)

Writes the workspace to a xml File.

Parameters

`workspace`: The workspace to be saved.
(type=view.workspace.CabelFrame)

Return Value

[True, ''] if saved, [False, 'Message'] if error.
(type=[boolean,string])

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.8.1.2 Instance Variables

Name	Description
doc	Root Document. *(type=xml.dom.Document)*
filePath	Path of the file to save. *(type=string)*
name	Name of the File (without extension '.cw'). *(type=string)*

A.9 Module model.xmlReader

A.9.1 Class ModuleDefinitionError

exceptions.Exception ─┐
ModuleDefinitionError

ModuleDefinitionError.

Exception if there is any syntax error in the module xml-file.

A.9.1.1 Methods

Inherited from Exception: __init__, __getitem__, __str__

A.9.2 Class ModuleNotFoundError

exceptions.Exception ─┐
ModuleNotFoundError

ModuleNotFoundError.

Exception if the searched Module wasn't found.

A.9.2.1 Methods

Inherited from Exception: __init__, __getitem__, __str__

A.9.3 Class XmlModuleReader

__builtin__.object ─┐
XmlModuleReader

XmlModuleReader.

Reads XML modules.

A.9.3.1 Methods

__init__(*self*)

Standard Constructor.

Overrides: __builtin__.object.__init__

getModule(*self*, *name*)

searches for the the xml-file name.xml in the searchpath directory and tries to instanciate a module object from it.

Parameters

`name`: Name of the Module we want to instanciate.
(type=str)

Return Value

Returns new module object.

Raises

`ModuleNotFoundError` If the Xml-File name.xml doesn't exist. @rtype : model.module.Module

`ModuleDefinitionError` If the Xml-File isn't valid

getModuleFromNode(*self*, *moduleNode*, *name*=`''`)

getModules(*self*, *tree*=`True`)

Return list of modules with their names.

Parameters

`tree`: Build tree of modules (recursive list of tuples).
(type=bool)

getModulesObjects(*self*, *tree*=`True`)

Return list of modules with realtive paths as names.

Parameters

`tree`: Build tree of modules (recursive list of tuples).
(type=bool)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.9.3.2 Instance Variables

Name	Description
searchPath	Path to directory with XML modules files. *(type=str)*

A.9.4 Class XmlWorkspaceReader

__builtin__.object
  └ **XmlWorkspaceReader**

XmlWorkspaceReader.

Reads Xml Workspace files (*.cw)

A.9.4.1 Methods

__init__(*self*, *fileLocation*, *controller*)

Standardconstructor.

Parameters

`fileLocation`: Path to a saved workspace file.
(type=string)

`controller`: CabelController.
(type=view.controller.CabelController)

Overrides: __builtin__.object.__init__

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.9.4.2 Instance Variables

Name	Description
additionalInfo	A dictionary mapping additionalInfo Parameter names to its values (string)
connections	A list of dictionaries with the keys: 'fromModuleId', 'fromVarId', 'toModuleId' and 'toVarId' for the connections saved in. Its value types are int. *(type=list)*
fileName	Name of the saved workspace file without extension .cw *(type=string)*

continued on next page

Name	Description
filePath	Path to the saved workpace file. *(type=string)*
instances	A dictionary mapping tuples (moduleName [string], moduleId [int]) to moduleInstance Nodes. *(type=dictionary.)*
modelModules	Dictionary mapping ids (int) to model modules in the saved workspace. *(type=dictionary)*
references	A dictionary mapping moduleNames to module Nodes with the same xml syntax as used for the module defintions. *(type=dictionary)*
rootNode	The root node of the saved workspace file. *(type=xml.dom.Node)*
viewModules	Dictionary mapping model modules to their corresponding view modules. *(type=dictionary)*

A.10 Module tools.config

A.10.1 Class Category

```
__builtin__.object ─┐
                    Category
```

Known Subclasses: Csound, Directories, ListVar, View

Category.

Abstract superclass for all Config Categories. If you want a coherent config.xml File do NOT use the Config_Var objects of vars[] directly!!! setVal/getVal are the methods to use!!

A.10.1.1 Methods

__init__(*self*, *name*, *configDoc*, *configXmlLocation*)

Standardconstructor.

Parameters

`name:`	Name of Category. *(type=string)*
`configDoc:`	config.xml DOM. *(type=minidom.Document)*
`configXmlLocation:`	Path to config.xml. *(type=string)*

Overrides: __builtin__.object.__init__

fillWithDefaultVars(*self*)

Fills vars[] with its Config_Var Objects. Has to be implemented by the non abstract inherited classes.

getDefault(*self*, *var*)

Gets the default of the Config_Var object in vars[var].

Parameters

`var`: index of Config_Var object in vars[]. Get it from the instance-constants of the implemented Category.
(type=int)

Return Value

Default value of var.
(type=object)

getEnumIndex(*self*, *var*)

Gets the enum index of the enumVar vars[var].

Parameters

`var`: index of Config_Var object in vars[]. Get it from the instance-constants of the implemented Category.
(type=int)

Raises

`IndexError` If the ConfigVar vars[var] isn't a ConfigEnumVar.

getVal(*self*, *var*)

Gets the Value of the Config_Var object in vars[var]. If it isn't already in the config.xml DOM, it is added and config.xml saved.

Parameters

`var`: index of Config_Var object in vars[]. Get it from the static-constants of the implemented Category.
(type=int)

Return Value

Value of var.
(type=object)

getVar(*self*, *var*)

insertVar(*self*, *index*, *var*)

Sets the ConfigVar var at the specified position in self.vars and fills the still unset vars with its index number.

Parameters

`index`: 0 based index of the var to add.
(type=int)

`var`: The ConfigVar to be added.
(type=config.ConfigVar)

setEnumIndex(*self*, *var*, *index*)

Sets the enum index of the enumVar vars[var] and saves it to the config.xml.

Parameters

`var`: index of Config_Var object in vars[]. Get it from the instance-constants of the implemented Category.
(type=int)

`index`: Index to which it should be set.
(type=int)

Raises

`IndexError` If the ConfigVar vars[var] isn't a ConfigEnumVar.

setVal(*self*, *var*, *val*)

Sets the Value of the Config_Var object in vars[var] and saves the config.xml DOM.

Parameters

`var`: index of Config_Var object in vars[]. Get it from the instance-constants of the implemented Category.
(type=int)

`val`: Value to which it should be set.
(type=string)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.10.1.2 Instance Variables

Name	Description
categoryNode	Node of the Category in config.xml DOM. *(type=minidom.Element)*
configDoc	config.xml DOM. *(type=minidom.Document)*

continued on next page

Name	Description
configXmlLocation	Path to config.xml. *(type=string)*
name	Name of Category. *(type=string)*
vars	List of ConfigVar objects. setVal/getVal are the methods to use for access to its Values!! *(type=list)*

A.10.2 Class Config

Config.

Encapsulates all Categories in the config file.

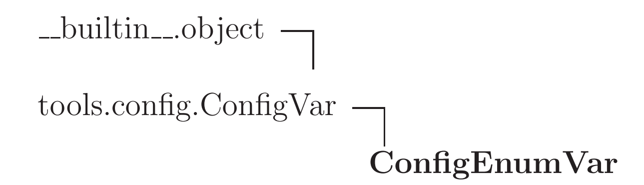
A.10.2.1 Methods

__init__(*self*)
Standardconstructor.

A.10.2.2 Instance Variables

Name	Description
configDoc	config.xml DOM. *(type=minidom.Document)*
configXmlLocation	Path to config.xml. *(type=string)*

A.10.3 Class ConfigEnumVar



ConfigEnumVar.

Like ConfigVar, but with additional instance var enum. Saves as the value the index of the value in the own enumeration list enum.

A.10.3.1 Methods

__init__(*self*, *category*, *name*, *default*, *setting*=`None`, *enum*=`[]`)

Standardconstructor.
Like ConfigVar constructor, with additional param enum.

Parameters

`default`: Index in self.enum
(type=int)

`enum`: List of predefined possible values to be selected.
(type=list)

Overrides: tools.config.ConfigVar.__init__

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__
Inherited from ConfigVar: getVal, read, setVal, write

A.10.3.2 Properties

Name	Description
Default	
Index	
Value	
Inherited from ConfigVar: setting *(p. 98)*	

A.10.3.3 Instance Variables

Name	Description
Default	Default Value (value of self.enum[self._default]).
enum	List of predefined possible values to be selected. *(type=list)*
Index	Actual index of ConfigEnumVar.
Value	Value of self.enum[self.Index]
Inherited from ConfigVar: category *(p. 98)*, description *(p. 98)*, name *(p. 98)*, nodeFound *(p. 98)*, nodeType *(p. 98)*, valueType *(p. 98)*	

A.10.4 Class ConfigVar

__builtin__.object
└ **ConfigVar**

Known Subclasses: ConfigEnumVar

ConfigVar.

Encapsulates information of a Config Variable and the functionality to write and read it.

!!! Always use the Property Value for accesing and setting the Value of ConfigVar!!!

A.10.4.1 Methods

__init__(*self*, *category*, *name*, *default*, *setting*=`None`, *nodeType*=`2`)

Standardconstructor.

Parameters

- `category`: Category to which ConfigVar belongs to.
  (type=config.Category)
- `name`: Name of ConfigVar.
  (type=string)
- `default`: Default-Value of ConfigVar.
  (type=string)
- `setting`: Describes ConfigVar.
  (type=config.Setting)
- `nodeType`: Is the ConfigVar stored as a ATTRIBUTE_NODE (=Standard) or TEXT_NODE in config.xml.
  (type=int (minidom.Node.nodeType))

Overrides: __builtin__.object.__init__

getVal(*self*)

Wrapper for category.getVal()

read(*self*)

Reads the Config Var out of config.xml. If it's var Node in config.xml DOM not exists, it adds it.

Return Value

Stored ConfigVar.
(type=ConfigVar)

setVal(*self*, *val*)

Wrapper for category.setVal()

write(*self*, *read*=`False`)

Writes Var in config.xml DOM. If its var Node does not exist, it creates it.

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.10.4.2 Properties

Name	Description
Default	
setting	
Value	

A.10.4.3 Instance Variables

Name	Description
category	Category to which ConfigVar belongs to. *(type=config.Category)*
description	Describes ConfigVar. *(type=string)*
name	Name of ConfigVar. *(type=string)*
nodeFound	Set by read method. If it is set to None after read method, the category in charge will save config.xml. Else it stores the varNode. *(type=boolean)*
nodeType	Is the ConfigVar stored as a ATTRIBUTE_NODE or TEXT_NODE in config.xml. *(type=int (minidom.Node.nodeType))*
valueType	Type of Value; 'string' *(type=string)*

A.10.5 Class Csound

__builtin__.object
tools.config.Category
Csound

config.Csound.

Category for Csound specific Information. (See documentation of superclass Category!)

A.10.5.1 Methods

__init__(*self*, *configDoc*, *configXmlLocation*)
Standardconstructor. **Parameters** `configDoc`: config.xml DOM. *(type=minidom.Document)* `configXmlLocation`: Path to config.xml. *(type=string)* Overrides: tools.config.Category.__init__

fillWithDefaultVars(*self*)
Fills classvar vars[] with its Config_Var Objects. Overrides: tools.config.Category.fillWithDefaultVars

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__
Inherited from Category: getDefault, getEnumIndex, getVal, getVar, insertVar, setEnumIndex, setVal

A.10.5.2 Instance Variables

Name	Description
Inherited from Category: categoryNode *(p. 94)*, configDoc *(p. 94)*, configXmlLocation *(p. 94)*, name *(p. 94)*, vars *(p. 94)*	

A.10.5.3 Class Variables

Name	Description
AUTOPLAY	**Value:** 7 *(type=int)*
CSOUNDPATH	**Value:** 5 *(type=int)*
FEEDBACK_TIMEOUT	**Value:** 8 *(type=int)*
KONTROLRATE	**Value:** 1 *(type=int)*
KSMPS	**Value:** 2 *(type=int)*
NCHNLS	**Value:** 3 *(type=int)*
PARAMS	**Value:** 6 *(type=int)*
SAMPLERATE	**Value:** 0 *(type=int)*
SCORE	**Value:** 4 *(type=int)*

A.10.6 Class Directories

__builtin__.object
tools.config.Category
Directories

Directories.

Category for Directories. (See documentation of superclass Category!)

A.10.6.1 Methods

__init__(*self*, *configDoc*, *configXmlLocation*)

Standardconstructor.

Parameters

`configDoc`: config.xml DOM.
(type=minidom.Document)

`configXmlLocation`: Path to config.xml.
(type=string)

Overrides: tools.config.Category.__init__

fillWithDefaultVars(*self*)

Fills classvar vars[] with its Config_Var Objects.

Overrides: tools.config.Category.fillWithDefaultVars

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__
Inherited from Category: getDefault, getEnumIndex, getVal, getVar, insertVar, setEnumIndex, setVal

A.10.6.2 Instance Variables

Name	Description
Inherited from Category: categoryNode *(p. 94)*, configDoc *(p. 94)*, configXmlLocation *(p. 94)*, name *(p. 94)*, vars *(p. 94)*	

A.10.6.3 Class Variables

Name	Description
EDITOR	**Value:** 4 *(type=`int`)*
LOGGING_DIR	**Value:** 2 *(type=`int`)*
LOGGING_ON	**Value:** 3 *(type=`int`)*
MODULES	**Value:** 0 *(type=`int`)*
RECENTFILES	**Value:** 1 *(type=`int`)*

A.10.7 Class ListVar

__builtin__.object
tools.config.Category
ListVar

List.

Helper category for internal use only. Serves as a meta category for list vars.

A.10.7.1 Methods

__init__(*self*, *configVar*, *list*=`[]`, *name*=`''`, *parentNode*=`None`, *setting*=`'take the 1 in configVar'`, *varNodeName*=`'var'`)

Standardconstructor.

Parameters

- `configVar:` ConfigVar of ListVar.
  (type=config.ConfigVar)
- `list:` The list of initial values.
  (type=list)
- `name:` Name of the list var.
  (type=string)
- `parentNode:` Parent Node for the ListVar to save.
  (type=xml.dom.Node)
- `setting:` Setting for the ListVar.
  (type=config.Setting)
- `varNodeName:` Name of the Node in which the ListVar is stored. 'var' per default!
  (type=string)

Overrides: tools.config.Category.__init__

fillWithDefaultVars(*self*)
Fills classvar vars[] with its Config_Var Objects. Overrides: tools.config.Category.fillWithDefaultVars

read(*self*)
Read the listVar from config.xml and return the obtained list. **Return Value** The listVar saved in config.xml *(type=list)*

write(*self*, *_list*=`None`)
Writes the as argument passed list to the config.xml. Therefore it deletes the actual Nodes and creates it new from self._list.

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__
Inherited from Category: getDefault, getEnumIndex, getVal, getVar, insertVar, setEnumIndex, setVal

A.10.7.2 Instance Variables

Name	Description
Inherited from Category: categoryNode *(p. 94)*, configDoc *(p. 94)*, configXmlLocation *(p. 94)*, name *(p. 94)*, vars *(p. 94)*	

A.10.8 Class Setting

__builtin__.object
  Setting

Setting.

Encapsulates information which serve for the graphical representation of the config.xml.

A.10.8.1 Methods

__init__(*self*, *description*, *displayName*)

Standardconstructor.

Parameters

`description`: Description of corresponding configVar. Used for Tooltip.
(type=string)

`displayName`: Display name of corresponding configVar.
(type=string)

Overrides: __builtin__.object.__init__

readNode(*self*, *settingNode*)

Read settingNode of corresponding configVar node in config.xml

Parameters

`settingNode`: Node in which the setting params are stored.
(type=xml.dom.minidom.Node)

updateView(*self*, *workSpace*)

Calls the update methods of all observers in model.workspace.

Parameters

`workSpace`: The Cabel view workspace.
(type=view.workspace.CabelFrame)

writeNode(*self*, *settingNode*)

Write settingNode to corresponding configVar node in config.xml

Parameters

`settingNode`: Node in which the setting params are stored.
(type=xml.dom.minidom.Node)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

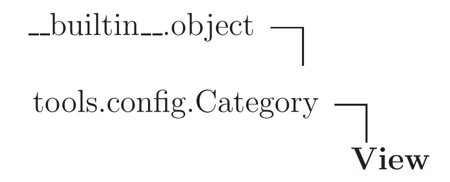
A.10.8.2 Properties

Name	Description
ctrlLength	

A.10.8.3 Instance Variables

Name	Description
choose	Specifies wether the configVar value can be choosen out of a control. Possible values are: 'file', 'path', or 'None' *(type=string)*
ctrlHeight	Number of lines for multiLine string configVar. *(type=int)*
description	Description of corresponding configVar. Used for Tooltip. *(type=string)*
dialog	Show ConfigVar in the preferences dialog? *(type=boolean)*
displayName	Display name of corresponding configVar. *(type=string)*
group	Corr. ConfigVar belongs should be displayed grouped in a box labeld with this string. ConfigVars with a group setting 'None' are displayed solemnly. *(type=string)*
hint	A link to a hint text fragment. *(type=string)*
notNone	Must not be None the value. *(type=boolean)*
updateFunc	If this is set, the configurator automatically notifies all in model.module.Workspace registered observers. The argument for the observers is the function name. *(type=string)*

A.10.9 Class View



View.

Category for graphic representations. (See documentation of superclass Category!)

A.10.9.1 Methods

__init__(*self*, *configDoc*, *configXmlLocation*)
Standardconstructor. **Parameters** `configDoc`: config.xml DOM. *(type=minidom.Document)* `configXmlLocation`: Path to config.xml. *(type=string)* Overrides: tools.config.Category.__init__

fillWithDefaultVars(*self*)
Fills classvar vars[] with its Config_Var Objects. Overrides: tools.config.Category.fillWithDefaultVars

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__
Inherited from Category: getDefault, getEnumIndex, getVal, getVar, insertVar, setEnumIndex, setVal

A.10.9.2 Instance Variables

Name	Description
Inherited from Category: categoryNode *(p. 94)*, configDoc *(p. 94)*, configXmlLocation *(p. 94)*, name *(p. 94)*, vars *(p. 94)*	

A.10.9.3 Class Variables

Name	Description
BACKGROUNDCOLOUR	**Value:** `0` *(type=`int`)*
BOTTOMWINDOW_ACTIVEPAGE	**Value:** `10` *(type=`int`)*
BOTTOMWINDOW_HEIGHT	**Value:** `11` *(type=`int`)*
BOTTOMWINDOW_REMEMBERPROPERTIES	**Value:** `9` *(type=`int`)*
BOTTOMWINDOW_SHOW	**Value:** `8` *(type=`int`)*

continued on next page

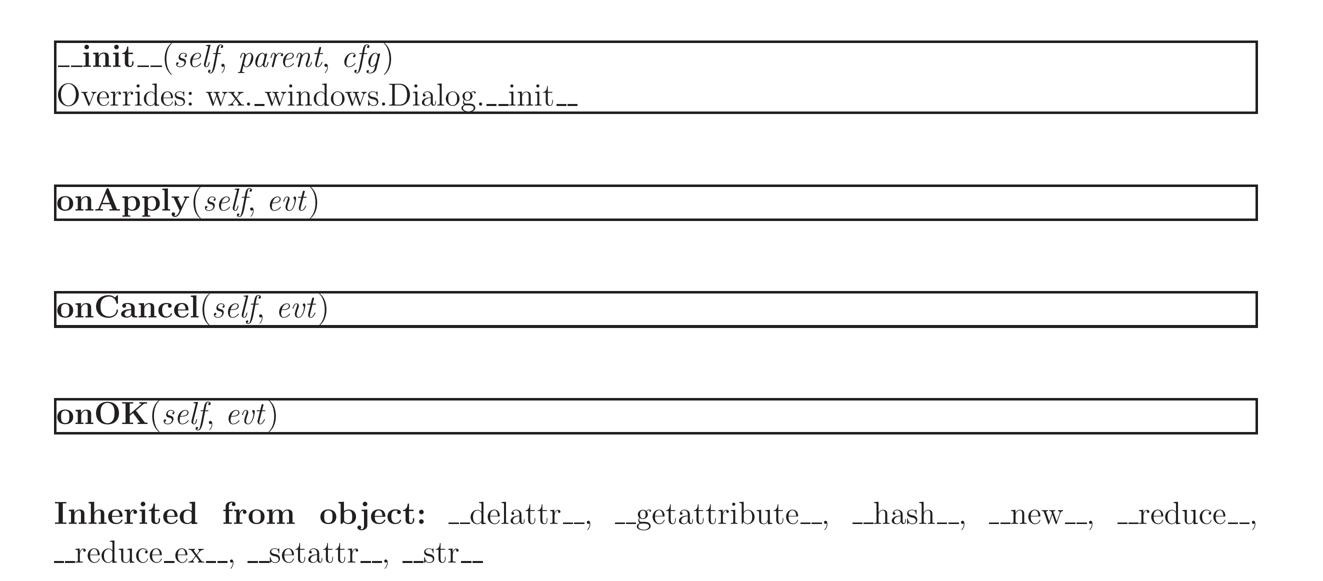
Name	Description
CABLECOLOUR	**Value:** 7 *(type=int)*
CABLESAGGING	**Value:** 12 *(type=int)*
FRAME_MAXIMIZED	**Value:** 15 *(type=int)*
FRAMEHEIGHT	**Value:** 4 *(type=int)*
FRAMEWIDTH	**Value:** 3 *(type=int)*
FULLMODULENAMES	**Value:** 5 *(type=int)*
MODULEDELETEWAR-NING	**Value:** 6 *(type=int)*
WORKSPACEHEIGHT	**Value:** 2 *(type=int)*
WORKSPACEWIDTH	**Value:** 1 *(type=int)*
ZOOM_FACTOR_DEFA-ULT	**Value:** 14 *(type=int)*
ZOOM_INDIVIDUAL_A-CTIVE	**Value:** 13 *(type=int)*
ZOOM_LASTVALUE	**Value:** 16 *(type=int)*

A.11 Module view.configurator

A.11.1 Class CabelConfigDialog

__builtin__.object
wx._core.Object
wx._core.EvtHandler
wx._core.Window
wx._windows.TopLevelWindow
wx._windows.Dialog
CabelConfigDialog

A.11.1.1 Methods



__init__(*self*, *parent*, *cfg*)
Overrides: wx._windows.Dialog.__init__

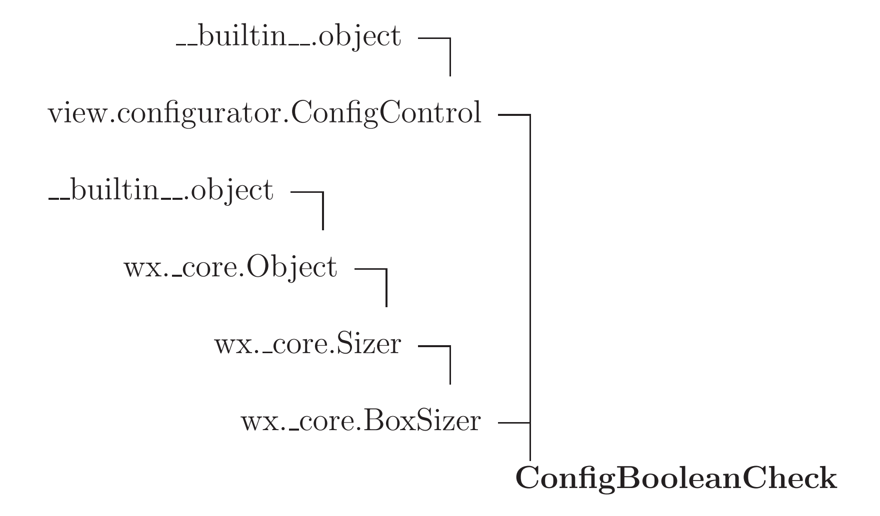
onApply(*self*, *evt*)

onCancel(*self*, *evt*)

onOK(*self*, *evt*)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __setattr__, __str__

A.11.2 Class ConfigBooleanCheck



A.11.2.1 Methods

__init__(*self*, *parent*, *id*, *configParser*, *configBooleanVar*, *size*)
Overrides: view.configurator.ConfigControl.__init__

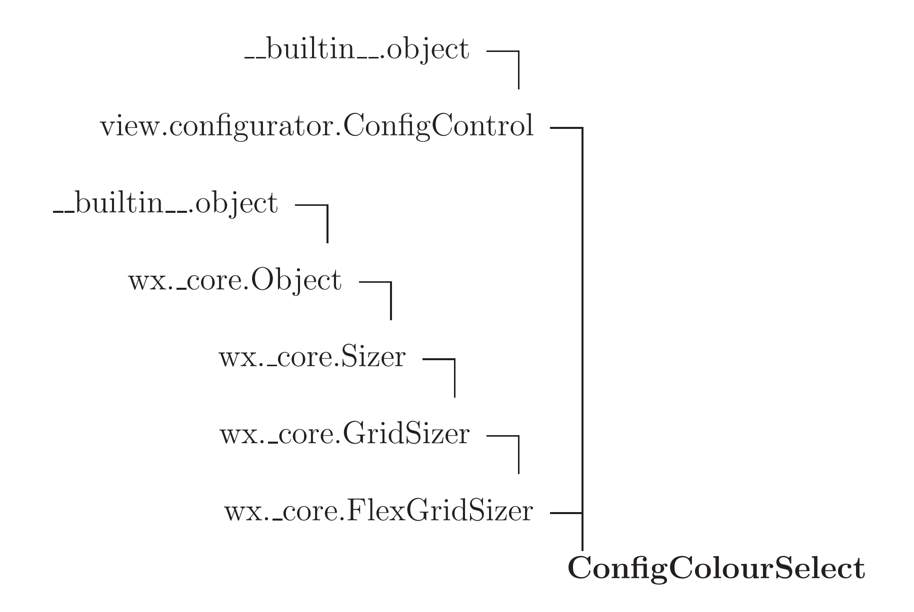
callback(*self*, *event*)
Virtual method of ConfigControl. Should call the updateVarValDict with the value of the control in order to process the input of the control. Overrides: view.configurator.ConfigControl.callback extit(inherited documentation)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __setattr__, __str__
Inherited from ConfigControl: hasChanged, invalidateVarVal, onOK, paintCtrlWhite, paintTheInvalidRed, updateVarValDict

A.11.2.2 Instance Variables

Name	Description
Inherited from ConfigControl: configParser *(p. 112)*, configVar *(p. 112)*, ctrl *(p. 112)*, parent *(p. 112)*	

A.11.3 Class ConfigColourSelect



A.11.3.1 Methods

__init__(*self*, *parent*, *id*, *configParser*, *configColourVar*, *size*, *label*='', *pos*=`wx.Point(-1, -1)`, *style*=0)
Overrides: view.configurator.ConfigControl.__init__

callback(*self*, *event*)
Virtual method of ConfigControl. Should call the updateVarValDict with the value of the control in order to process the input of the control.
Overrides: view.configurator.ConfigControl.callback extit(inherited documentation)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __setattr__, __str__
Inherited from ConfigControl: hasChanged, invalidateVarVal, onOK, paintCtrlWhite, paintTheInvalidRed, updateVarValDict

A.11.3.2 Instance Variables

Name	Description
Inherited from ConfigControl: configParser *(p. 112)*, configVar *(p. 112)*, ctrl *(p. 112)*, parent *(p. 112)*	

A.11.4 Class ConfigControl

__builtin__.object
└ **ConfigControl**

Known Subclasses: ConfigBooleanCheck, ConfigColourSelect, ConfigFloat, ConfigInt, ConfigStringMultiLine, ConfigStringSingleLine

ConfigControl.

Controller for the configurator input controlls.

It's an 'abstract' class. Inherited classes can implement 'abstract' invalidateVarVal(self, val) for validation of the input to the control.

A.11.4.1 Methods

__init__(*self*, *parent*, *configParser*, *var*)

Standardconstructor.

Parameters

`parent`: The root panel of a wx.Notebook (category) page.
(type=wx.Panel)

`configParser`: ConfigParser in charge of this control.
(type=configurator.ConfigParser)

`var`: ConfigVarrepresented through this control.
(type=config.ConfigVar)

Overrides: __builtin__.object.__init__

callback(*self*, *event*)

Virtual method of ConfigControl. Should call the updateVarValDict with the value of the control in order to process the input of the control.

hasChanged(*self*, *val*)

Checks if the value of the configVarhas changed.

Parameters

`val`: The value returned by the control.
(type=self.configVar.valueType)

invalidateVarVal(*self*, *val*)

Method for invalidation of the configVar's value.

Return Value

A warning message if the Value is invalid, else False.
(type=boolean/string)

onOK(*self*, *event*)

Calls the callback method of the CabelConfig control if the accelerator for the OK or Apply Buttons (ALT-O or ALT-A) are pressed.

paintCtrlWhite(*self*, *event*)

Paints the control as it was originally.

paintTheInvalidRed(*self*)

Paints the control red.

updateVarValDict(*self*, *val*)

Checks if the value val for the ConfigVar has changed, if it is valid and caches the configVar/value pair in the responsible configParser.

Parameters

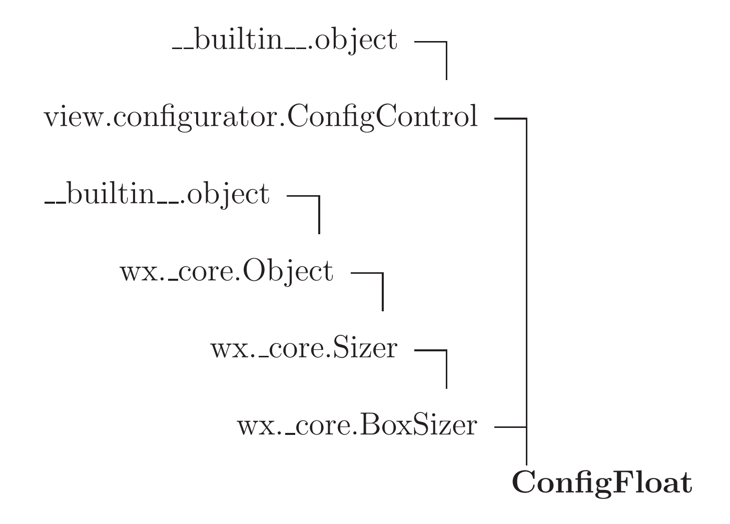
`val`: The value returned by the control.
(type=self.configVar.valueType)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.11.4.2 Instance Variables

Name	Description
configParser	ConfigParser in charge of this control. *(type=configurator.ConfigParser)*
configVar	ConfigVarrepresented through this control. *(type=tools.config.ConfigVar)*
ctrl	The input control. has to be set in the constructor of inherited class. *(type=wx.Control)*
parent	The root panel of a wx.Notebook (category) page. *(type=wx.Panel)*

A.11.5 Class ConfigFloat



A.11.5.1 Methods

__init__(*self*, *parent*, *id*, *configParser*, *configFloatVar*, *_size*)
Overrides: view.configurator.ConfigControl.__init__

callback(*self*, *event*)

Virtual method of ConfigControl. Should call the updateVarValDict with the value of the control in order to process the input of the control.

Overrides: view.configurator.ConfigControl.callback extit(inherited documentation)

invalidateVarVal(*self*, *val*)

Method for invalidation of the configVar's value.

Return Value

A warning message if the Value is invalid, else False.
(type=boolean/string)

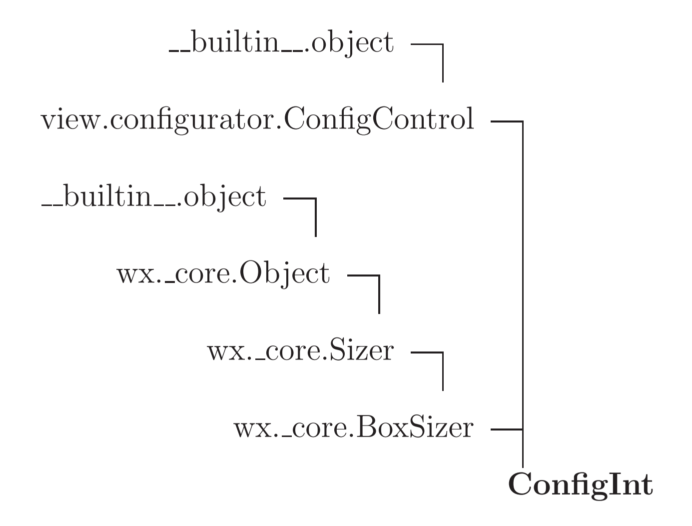
Overrides: view.configurator.ConfigControl.invalidateVarVal extit(inherited documentation)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __setattr__, __str__
Inherited from ConfigControl: hasChanged, onOK, paintCtrlWhite, paintTheInvalidRed, updateVarValDict

A.11.5.2 Instance Variables

Name	Description
Inherited from ConfigControl: configParser *(p. 112)*, configVar *(p. 112)*, ctrl *(p. 112)*, parent *(p. 112)*	

A.11.6 Class ConfigInt



A.11.6.1 Methods

__init__(*self*, *parent*, *id*, *configParser*, *configIntVar*, *_size*)
Overrides: view.configurator.ConfigControl.__init__

callback(*self*, *event*)
Virtual method of ConfigControl. Should call the updateVarValDict with the value of the control in order to process the input of the control. Overrides: view.configurator.ConfigControl.callback extit(inherited documentation)

invalidateVarVal(*self*, *val*)
Method for invalidation of the configVar's value. **Return Value** A warning message if the Value is invalid, else False. *(type=boolean/string)* Overrides: view.configurator.ConfigControl.invalidateVarVal extit(inherited documentation)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __setattr__, __str__
Inherited from ConfigControl: hasChanged, onOK, paintCtrlWhite, paintTheInvalidRed, updateVarValDict

A.11.6.2 Instance Variables

Name	Description
Inherited from ConfigControl: configParser *(p. 112)*, configVar *(p. 112)*, ctrl *(p. 112)*, parent *(p. 112)*	

A.11.7 Class ConfigParser

```
__builtin__.object ─┐
                    ConfigParser
```

A.11.7.1 Methods

__init__(*self*, *config*, *parent*, *worksp*)

Standardconstructor.

Parameters

`config`: The config model to be parsed.
(type=tools.config.Config)

Overrides: __builtin__.object.__init__

fillCategoryNotebook(*self*)

getBooleanVarCtrl(*self*, *parent*, *var*, *id*=`-1`, *size*=`wx.Size(-1, -1)`)

getCategoryPanel(*self*, *category*)

getColourVarCtrl(*self*, *parent*, *var*, *id*=`-1`, *size*=`wx.Size(60, 20)`)

getControl(*self*, *var*, *parent*)

getEnumVarCtrl(*self*, *parent*, *var*)

getFloatVarCtrl(*self*, *parent*, *var*, *id*=`-1`, *size*=`wx.Size(-1, -1)`)

getGroupPanel(*self*, *vars*, *parent*)

getIntVarCtrl(*self*, *parent*, *var*, *id*=`-1`, *size*=`wx.Size(-1, -1)`)

getListVarCtrl(*self*, *parent*, *var*)

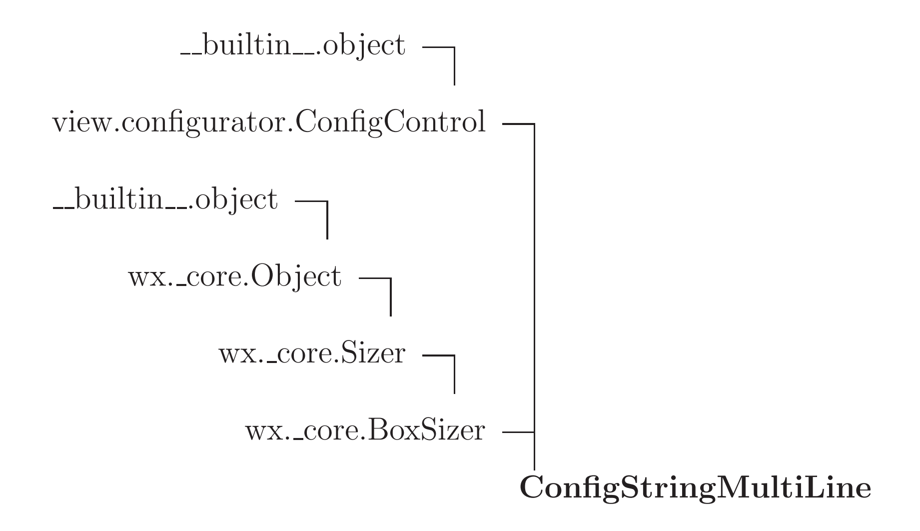
getStringVarMultiLineCtrl(*self*, *parent*, *var*, *id*=`-1`, *size*=`wx.Size(-1, -1)`)

getStringVarSingleLineCtrl(*self*, *parent*, *var*, *id*=`-1`, *size*=`wx.Size(-1, -1)`)

SaveVars(*self*)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.11.8 Class ConfigStringMultiLine



A.11.8.1 Methods

__init__(*self*, *parent*, *id*, *configParser*, *configStringTextNodeVar*, *_size*)
Overrides: view.configurator.ConfigControl.__init__

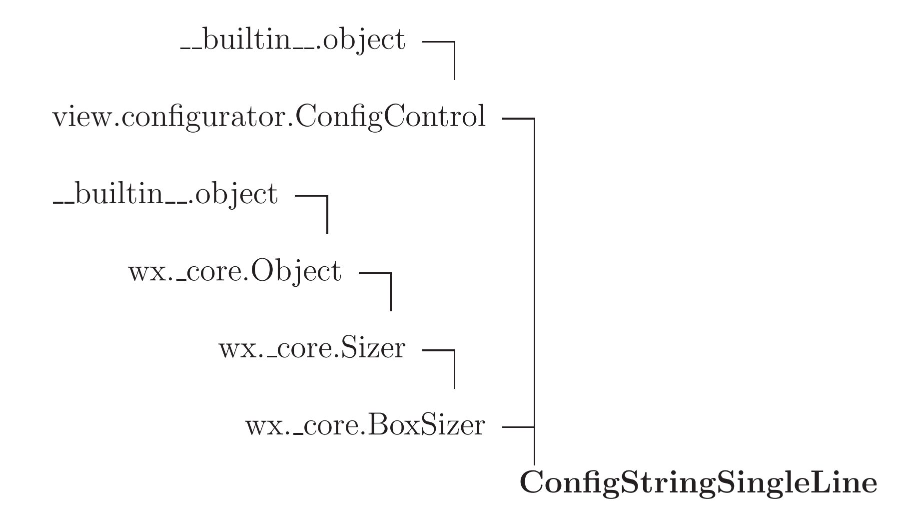
callback(*self*, *event*)
Virtual method of ConfigControl. Should call the updateVarValDict with the value of the control in order to process the input of the control.
Overrides: view.configurator.ConfigControl.callback extit(inherited documentation)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __setattr__, __str__
Inherited from ConfigControl: hasChanged, invalidateVarVal, onOK, paintCtrlWhite, paintTheInvalidRed, updateVarValDict

A.11.8.2 Instance Variables

Name	Description
Inherited from ConfigControl: configParser *(p. 112)*, configVar *(p. 112)*, ctrl *(p. 112)*, parent *(p. 112)*	

A.11.9 Class ConfigStringSingleLine



A.11.9.1 Methods

__init__(*self*, *parent*, *id*, *configParser*, *configStringTextNodeVar*, *_size*) Overrides: view.configurator.ConfigControl.__init__

callback(*self*, *event*)
Virtual method of ConfigControl. Should call the updateVarValDict with the value of the control in order to process the input of the control. Overrides: view.configurator.ConfigControl.callback extit(inherited documentation)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __setattr__, __str__
Inherited from ConfigControl: hasChanged, invalidateVarVal, onOK, paintCtrlWhite, paintTheInvalidRed, updateVarValDict

A.11.9.2 Instance Variables

Name	Description
Inherited from ConfigControl: configParser *(p. 112)*, configVar *(p. 112)*, ctrl *(p. 112)*, parent *(p. 112)*	

A.12 Module view.connection

A.12.1 Class Connection

__builtin__.object
└ **Connection**

Known Subclasses: ModuleConnection

Connection.

Graphic class for connection from start point to end point.

A.12.1.1 Methods

__init__(*self*, *startPt*, *endPt*)

Standard constructor.

Parameters

`startPt`: Start point of connection.
(type=wx.Point)

`endPt`: End point of connection.
(type=wx.Point)

Overrides: __builtin__.object.__init__

draw(*self*, *dc*, *zoom*=`100`)

Draw this connection in our workspace.

Parameters

`dc`: Device conetext on which to draw.
(type=wx.DC)

`zoom`: Zoom of Connection; default = 100.
(type=int)

drawRelative(*self*, *dc*, *origin*, *zoom*=100)

Draw this connection relative to origin in our workspace.

Parameters

`dc`: Device context on which to draw.
(type=wx.DC)

`origin`: Actual orgin of workspace.
(type=wx.Point)

`zoom`: Zoom of Connection; default = 100.
(type=int)

getColours(*self*)

Calculates the 3 colours of a cabel connection out of the basic value in config.xml

Return Value

A dictionary with the entries: 'shade', 'midtone' and 'highlight'
(type=dict of wx.colour)

getSagging(*self*)

Get the sagging of connection cables as saved in config.xml.

Return Value

Sagging of cable connections.
(type=int)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.12.1.2 Instance Variables

Name	Description
config	View related config vars. *(type=tools.config.View)*
endPt	End Point of connection. *(type=wx.Point)*
startPt	Start point of connection. *(type=wx.Point)*

A.12.2 Class ModuleConnection

__builtin__.object
 └ view.connection.Connection
  └ **ModuleConnection**

Graphic class for module connections.

A.12.2.1 Methods

__init__(*self*, *outModule*, *outputNum*, *inModule*, *inputNum*)

Standard constructor.

Parameters

- `outModule`: Start module of connection.
  (type=view.module.Module)
- `outputNum`: Output number of start plug on outModule.
  (type=int)
- `inModule`: End module of connection.
  (type=view.module.Module)
- `inputNum`: Input number of end plug on inModule.
  (type=int)

Overrides: view.connection.Connection.__init__

__eq__(*self*, *other*)

Overwritten equal operator.

Parameters

- `other`: the ModuleConnection to compare with
  (type=view.connection.ModuleConnection)

__ne__(*self*, *other*)

Overwritten not equal operator.

Parameters

- `other`: the ModuleConnection to compare with
  (type=view.connection.ModuleConnection)

draw(*self*, *dc*, *zoom*)

Draw this connection in our workspace.

Parameters

`dc`: Device conetext on which to draw.
(type=wx.DC)

`zoom`: Zoom of Connection; default = 100.
(type=int)

Overrides: view.connection.Connection.draw

drawRelative(*self*, *dc*, *origin*, *zoom*)

Draw this connection relative to origin in our workspace.

Parameters

`dc`: Device context on which to draw.
(type=wx.DC)

`origin`: Actual orgin of workspace.
(type=wx.Point)

`zoom`: Zoom of Connection; default = 100.
(type=int)

Overrides: view.connection.Connection.drawRelative

getInModule(*self*)

Return input moduleof connection.

getInputNumber(*self*)

Return input number of inModule.

getOutModule(*self*)

Return output module of connection.

getOutputNumber(*self*)

Return output number of outModule.

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__
Inherited from Connection: getColours, getSagging

A.12.2.2 Instance Variables

Name	Description
inModule	End module of connection. *(type=view.module.Module)*
inputNum	Input number of end plug on inModule. *(type=int)*
outModule	Start module of connection. *(type=view.module.Module)*
outputNum	Output number of start plug on outModule. *(type=int)*
Inherited from Connection: config *(p. 119)*, endPt *(p. 119)*, startPt *(p. 119)*	

A.13 Module view.controller

A.13.1 Class CabelController

```
__builtin__.object ─┐
                    │
model.observer.Observer ─┐
                         │
                  CabelController
```

CabelController.

Controller and event handling for CabelFrame.

A.13.1.1 Methods

__init__(*self*, *model*, *view*)

Standard constructor.

Parameters

`model`: Corresponding model for this controller.
(type=model.workspace.Workspace)

`view`: Corresponding view for this controller.
(type=view.workspace.CabelFrame)

Overrides: model.observer.Observer.__init__

onClose(*self*, *event*)

Respond to the Frame close event.

Parameters

`event`: *(type=)*

onKey(*self*, *event*)

Respond to key events.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onMenuExit(*self*, *event*)

Respond to the Ëxitmenu command.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onMenuExportToCsd(*self*, *event*)

Respond to Ëxport to CSDmenu command.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onModulesMenu(*self*, *event*)

Listener for the Module menu.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onMouseLeftDclick(*self*, *event*)

Respond to left mouse button double click on workspace.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onMouseLeftDown(*self*, *event*)

Respond to left mouse button down on workspace.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onMouseLeftUp(*self*, *event*)

Respond to left mouse button up on workspace.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onMouseMiddleDown(*self*, *event*)

Respond to middle mouse button down on workspace.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onMouseMotion(*self*, *event*)

Respond to mouse motion on workspace.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onMouseRightDown(*self*, *event*)

Respond to right mouse button down on workspace.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onNew(*self*, *event*)

Respond to the Newmenu command.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onOpen(*self*, *event*)

Respond to the Öpenmenu command.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onOptionsOpen(*self*, *event*)

Open the Options Dialog.

Parameters

`event`: Event.
(type=wx.Event)

onPlayStop(*self*, *event*)

EventHandler for start/stop csound process from the file menu or with shortcut ctrl-y

Parameters

`event`: Menu Event associated with this method in view.workspace.CabelFrame.__init__
(type=wx.Event)

onRecent(*self*, *event*)

Open file from recent open files menu.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onRemoveActModule(*self*, *event*)

Remove a module. If VIEW_MODULEDELETEWARNING is enabled ask before removing.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onSave(*self*, *event*)

Respond to the SSavemenu command.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onSaveAs(*self*, *event*)

Respond to the SSave Asmenu command.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

onScaleInActModule(*self*, *event*)

Scales into (Zoom In) the selected Module with default zoom from preferences.

Parameters

`event`: Menu Event associated with this method in view.workspace.CabelFrame.getModuleMenu
(type=wx.Event)

onScaleOutActModule(*self*, *event*)

Scales out of (Zoom Out) the selected Module with default zoom from preferences.

Parameters

`event`: Menu Event associated with this method in view.workspace.CabelFrame.getModuleMenu
(type=wx.Event)

onShowModuleXML(*self*, *event*)

Opens the Module Xml file in editor.

Parameters

`event`: Menu Event associated with this method in view.workspace.CabelFrame.getModuleMenu
(type=wx.Event)

onSize(*self*, *event*)

Saves wether view.workspace.CabelFrame is maximized or not in the config.xml

Parameters

`event`: Menu Event associated with this method in view.workspace.CabelFrame.__init__
(type=wx.Event)

onValueFrameClosed(*self*, *valueframe*)

Called when a value frame is closed.

Parameters

`valueframe`: Closed value frame.
(type=model.view.valueframe.CabelValueFrame)

setIoTextCtrl(*self*, *control*=`None`)

setModuleFocus(*self*, *module*)

Sets the actual Module, gives it a focus and removes the focus on the old actual module if there was any.

Parameters

`module`: The new actual Module.
(type=view.module.Module)

update(*self*, *observable*, *arg*)

This method is called whenever the observed object is changed. An application calls an observable object's notifyObservers method to have all the object's observers notified of the change.

Parameters

`arg`: An argument passed to the notifyObservers method.
(type=object)

Overrides: model.observer.Observer.update

zoom(*self*, *zoom*)

Set Zoom percent-value on the workspace, scales every module on the workspace and repaints the workspace.

Parameters

`zoom`: Percent value of the zoom
(type=int)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.14 Module view.module

A.14.1 Class Module

```
__builtin__.object ─┐
                    Module
```

Module.

Graphic class for a module on the workspace.

A.14.1.1 Methods

__init__(*self, x, y, module, controller*)

Standard constructor.

Parameters

`x`: X coordintate of module on workspace.
(type=wx.Point)

`y`: Y coordintate of module on workspace.
(type=wx.Point)

`module`: Corresponding model.module.Module.
(type=model.module.Module)

`controller`: CabelController.
(type=view.controller.CabelController)

Overrides: __builtin__.object.__init__

draw(*self, dc*)

Draw this module on workspace.

Parameters

`dc`: Device context on which to draw the module bitmap.
(type=wx.DC)

drawRelative(*self, dc, origin*)

Draw this module relative to origin in our workspace.

Parameters

`dc`: Device context on which to draw the module bitmap.
(type=wx.DC)

`origin`: Actual origin of workspace.
(type=wx.Point)

getInput(*self*, *num*)

Returns point of input plug.

Parameters

`num`: Number of input. @rtype : wx.Point
(type=int)

Return Value

Point of input.

getInVar(*self*, *num*)

Returns inVar of input num.

Parameters

`num`: Number of output. @rtype : model.var.InVar
(type=int)

Return Value

InVar.

getName(*self*)

Return name of module.

Return Value

Name of module.
(type=string)

getOutput(*self*, *num*)

Returns point of output plug.

Parameters

`num`: Number of output. @rtype : wx.Point
(type=int)

Return Value

Point of output.

getOutVar(*self*, *num*)

Returns OutVar of output num.

Parameters

`num`: Number of output. @rtype : model.var.OutVar
(type=int)

Return Value

OutVar.

isOnInput(*self*, *pt*)

If point is on an input returns the input number. Otherwise it returns -1.

Parameters

`pt`: Test if pf is on an input. @rtype : int
(type=wx.Point)

Return Value

Number of input on which pt is (0 indexed). Otherwise -1.

isOnModule(*self*, *pt*)

Returns True if this module contains given point.

Parameters

`pt`: Test if pt is included in module. @rtype : bool
(type=wx.Point)

Return Value

Is pt on module?

isOnOutput(*self*, *pt*)

If point is on an output returns the output number. Otherwise it returns -1.

Parameters

`pt`: Test if pf is on an output. @rtype : int
(type=wx.Point)

Return Value

Number of output on which pt is (0 indexed). Otherwise -1.

refresh(*self*)

scaleIt(*self*, *scaleFactor*)

setPosition(*self*, *pt*)

Set position of module.

Parameters

`pt`: New position of module.
(type=wx.Point)

setRelativePosition(*self*, *vecPt*)

Move module in vector direction.

Parameters

`vecPt`: Direction vector for movement of module.
(type=wx.Point)

zoom(*self*)
Repaint module zoomed.

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.14.1.2 Properties

Name	Description
scale	

A.14.1.3 Instance Variables

Name	Description
config	Config for some vars. *(type=tools.config.Config)*
controller	CabelController. *(type=view.controller.CabelController)*
height	Height in pixels. *(type=int)*
module	Corresponding model.module.Module. *(type=model.module.Module)*
numInputs	Number of inputs. *(type=int)*
numOutputs	Number of outputs. *(type=int)*
scale	Scale factor of module; Read-only property; considers zoom.
width	Width in pixels. *(type=int)*
x	X coordintate of module on workspace. *(type=wx.Coord)*
y	Y coordintate of module on workspace. *(type=wx.Coord)*

A.15 Module view.valueframe

A.15.1 Class CabelSlider

__builtin__.object
  └─ **CabelSlider**

This is a combination of wx.Slider and wx.TextCtrl in one horizontal Sizer.

A.15.1.1 Methods

__init__(*self*, *parent*, *inVar*, *model*)

Standard constructor.

Parameters

`inVar`: Slider/SpinCtrl controls this inVar.
(type=model.var.InVar)

Overrides: __builtin__.object.__init__

disable(*self*)

Disables slider and text ctrl.

enable(*self*)

Enables slider and text ctrl.

getSizer(*self*)

Return sizer for CabelSizer.

onSliderChange(*self*, *event*)

Called when slider is changed.

onTextChange(*self*, *event*)

Called when text ctrl is changed.

setSliderFocus(*self*)

Set focus to slider widget.

setTextFocus(*self*)

Set focus to text widget.

setToValue(*self*, *value*)

Sets slider and text ctrl to value.

Parameters

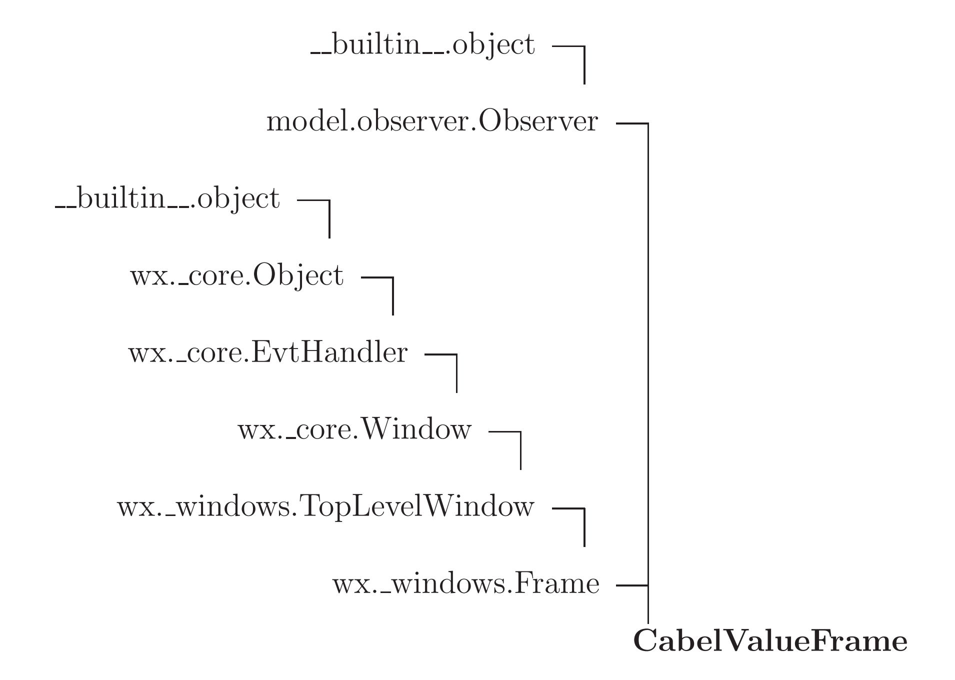
`value`: New value for slider and text ctrl.
(type=float)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.15.1.2 Instance Variables

Name	Description
inVar	Slider/SpinCtrl controls this inVar. *(type=model.var.InVar)*
parent	Parent window for CabelSlider. *(type=wx.Window)*

A.15.2 Class CabelValueFrame



CabelValueFrame.

Frame to show input values of a model module and sliders to change those values.

A.15.2.1 Methods

__init__(*self*, *module*, *parent*, *model*, *controller*, *position*=`(0, 0)`)

Standard constructor.

Parameters

`module:` This frame can change module's inVars.
(type=model.module.Module)

`parent:` Parent of CabelValueFrame (always on top of parent).
(type=wx.Window)

`model:` Corresponding model for this frame.
(type=model.workspace.Workspace)

`controller:` *(type=model.controller.CabelController)*

`position:` Initial position of frame.
(type=tuple)

Overrides: wx._windows.Frame.__init__

onClose(*self*, *event*)

Called when closing value frame.

onKey(*self*, *event*)

Called when getting a key event input.

update(*self*, *observable*, *arg*)

This method is called whenever the observed object is changed. An application calls an observable object's notifyObservers method to have all the object's observers notified of the change.

Parameters

`arg:` An argument passed to the notifyObservers method.
(type=object)

Overrides: model.observer.Observer.update

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __setattr__, __str__

A.16 Module view.workspace

A.16.1 Class CabelBottomWindow

```
__builtin__.object
    wx._core.Object
        wx._core.EvtHandler
            wx._core.Window
                wx._core.Control
                    wx._controls.BookCtrlBase
                        wx._controls.Notebook
                            CabelBottomWindow
```

Cabel's bootom area.

A.16.1.1 Methods

__init__(*self*, *parent*, *size*)

Standardconstructor.

Parameters

`parent`: Parent frame.
(type=view.workspace.CabelSplitterWindow)

`size`: The initial size of the bottom window pane
(type=wx.Size)

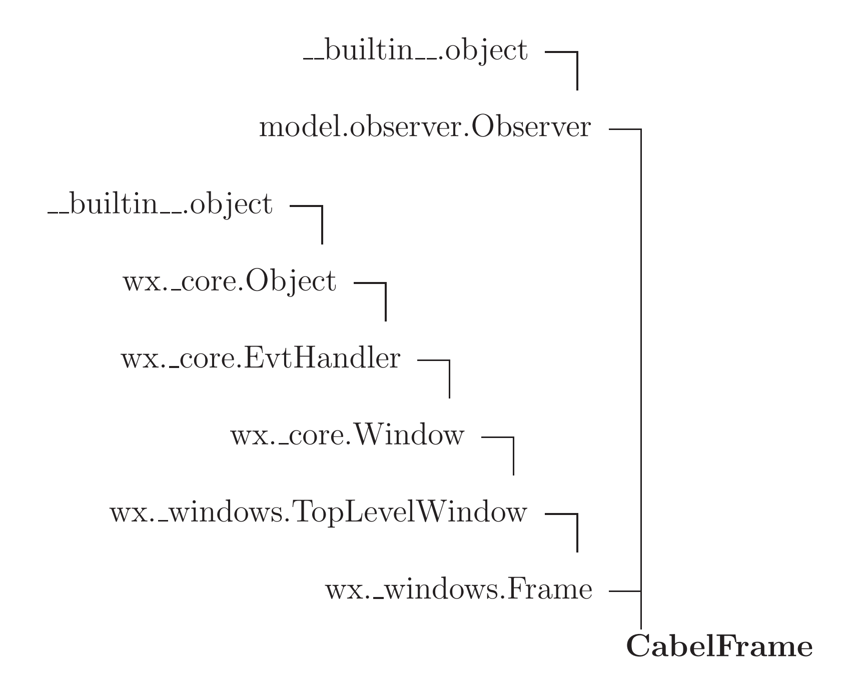
Overrides: wx._controls.Notebook.__init__

onPageChanged(*self*, *event*)

Save actual Page.

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __setattr__, __str__

A.16.2 Class CabelFrame



CabelFrame.

A frame showing the contents of a single Cabel session.

A.16.2.1 Methods

__init__(*self*, *model*)

Standard constructor.

Parameters

`model`: Corresponding model for this view.
(type=model.workspace.Workspace)

Overrides: wx._windows.Frame.__init__

addConnection(*self*, *connection*)

Add connection to view workspace.

Parameters

`connection`: Model connection to add to view workspace.
(type=model.connection.Connection)

addModule(*self*, *module*, *pt*)

Adds module to view workspace.

Parameters

`module`: module to add to view workspace
(type=model.module.Module)

`pt`: Position on the workspace of module to add
(type=wx.Point)

createDraggedCable(*self*, *startPt*, *endPt*)

Create connection between startPt and endPt as visual feedback for connection mode.

Parameters

`startPt`: Start point of dragged cable.
(type=wx.Point)

`endPt`: End point of dragged cable.
(type=wx.Point)

createWorkspace(*self*, *workspaceReader*)

getConnections(*self*)

Return list of view connections.

Return Value

List of view connections.
(type=List)

getDraggedCable(*self*)

Return dragged cable object.

Return Value

Visual feedback when connection two modules.
(type=model.view.connection.Connection)

getModuleAt(*self*, *pt*)

Return the first module found at given point.

Parameters

`pt`: Point to test if on a module. @rtype : view.module.Module
(type=wx.Point)

Return Value

Module at point or None.

getModuleMenu(*self*)

Return menu when right clicking on a module.

getModules(*self*)

Return list of view modules.

Return Value

List of view modules.
(type=List)

getModulesMenu(*self*, *xmlModuleList*)

Gets the Menu for all the defined Xml Modules.

Parameters

`xmlModuleList`: List of tuples which represents the structure of the modules folder.
(type=list)

Return Value

Menu
(type=wx.Menu)

GetTitle(*self*)

Return Title string for the CabelFrame.

Return Value

Title string for the CabelFrame.
(type=string)

Overrides: wx._windows.TopLevelWindow.GetTitle

onToggleBottomPane(*self*, *event*)

Toggle bottom pane in the splitter window.

Parameters

`event`: Event associated with this function.
(type=wx.Event)

reloadRecentMenu(*self*)

removeConnection(*self*, *connection*)

Remove connection from view workspace.

Parameters

`connection`: Connection to remove from view workspace.
(type=model.connection.Connection)

removeDraggedCable(*self*)

Remove dragged cable from workspace.

removeModule(*self*, *module*, *connections*)

Removes module from workspace.

Parameters

`module`: module to be removed from view workspace.
(type=model.module.Module)

`connections`: list of model.connection.Connection to delete.
(type=list)

scrollWorkspaceOnBorder(*self*, *pt*)

If pt is on workspace border scrolls workspace.

Parameters

`pt`: Point to test if on worspace border.
(type=wx.Point)

update(*self*, *observable*, *arg*)

This method is called whenever the observed object is changed. An application calls an observable object's notifyObservers method to have all the object's observers notified of the change.

Parameters

`arg`: An argument passed to the notifyObservers method.
(type=object)

Overrides: model.observer.Observer.update

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __setattr__, __str__

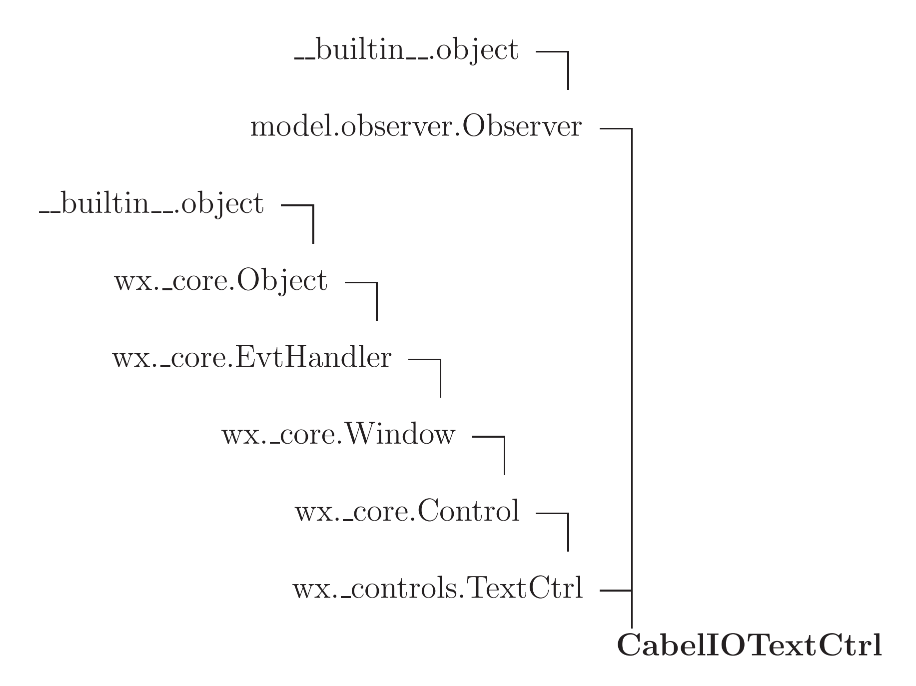
A.16.2.2 Instance Variables

Name	Description
config	relevant config Vars *(type=tools.config.Config.View)*
fileName	Name of the workspace *(type=string)*
filePath	Save-Path of the workspace. *(type=string)*
ioTextCtrl	Gets the stdout and stderr output of the app *(type=wx.TextCtrl)*

continued on next page

Name	Description
statusbar	Statusbar of CabelFrame *(type=wx.StatusBar)*
workspace	Workspace for synth building. *(type=view.workspace.CabelScrolledWindow)*
zoom	Zoom. *(type=int)*

A.16.3 Class CabelIOTextCtrl



CabelIOTextCtrl.

TextControl to which all the stderr/stdout messages will be written.

A.16.3.1 Methods

__init__(*self*, *parent*, *controller*)

Standardconstructor.

Parameters

`parent`: Parent on which the textControl should be put on.
(type=wx.Panel)

`controller`: Cabel controller with link to the instances of model/view.workspace.
(type=view.controller.CabelController)

Overrides: wx._controls.TextCtrl.__init__

update(*self*, *observable*, *arg*)

This method is called whenever the observed object is changed. An application calls an observable object's notifyObservers method to have all the object's observers notified of the change.

Parameters

`arg`: An argument passed to the notifyObservers method.
(type=object)

Overrides: model.observer.Observer.update

write(*self*, *txt*)

Write given text to the TextControl and to a logging file, if logging is set on.

Parameters

`txt`: Text to be written.
(type=string)

Overrides: wx._controls.TextCtrl.write

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __setattr__, __str__

A.16.3.2 Instance Variables

Name	Description
controller	Cabel controller with link to the instances of model/view.workspace. *(type=view.controller.CabelController)*
log	The logging file to which the logging messages are written or Null. *(type=File)*
loggingConfig	The directory category of config.xml in which the logging properties are saved. *(type=tools.config.Directories)*
loggingOn	State of logging. *(type=boolean)*

A.16.4 Class CabelScrolledWindow

```
__builtin__.object
  └─ wx._core.Object
       └─ wx._core.EvtHandler
            └─ wx._core.Window
                 └─ wx._windows.Panel
                      └─ wx._windows.ScrolledWindow
                           └─ CabelScrolledWindow
```

CabelScrolledWindow.

Scrolled window to represent the Cabel GUI workspace for placing and connecting modules with double buffered painting.

A.16.4.1 Methods

__init__(*self*, *parent*, *id*, *view*)

Standard constuctor.

Parameters

`parent`: Parent of this scrolled window.
(type=wx.Object)

`id`: ID of this scrolled window.
(type=int)

`view`: Corresponding view.
(type=view.workspace.CabelFrame)

Overrides: wx._windows.ScrolledWindow.__init__

GetBackgroundColour(*self*)

Overwrites wx.ScrolledWindow.GetBackgroundColour() with the in config.xml defined Colour.

Return Value

Background colour for workspace.
(type=wx.Colour)

Overrides: wx._core.Window.GetBackgroundColour

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __setattr__, __str__

A.16.5 Class CabelSplitterWindow

```
__builtin__.object ─┐
        wx._core.Object ─┐
        wx._core.EvtHandler ─┐
                wx._core.Window ─┐
         wx._windows.SplitterWindow ─┐
                                     CabelSplitterWindow
```

CabelSplitterWindow.

Horizontal splitter window which expands its top window when resized.

A.16.5.1 Methods

__init__(*self*, *parent*, *id*)

Standard constructor.

Parameters

`parent:` Parent of this splitter window.
(type=view.workspace.CabelFrame)

`id:` ID of this splitter window.
(type=int)

Overrides: wx._windows.SplitterWindow.__init__

getBottomWindowHeight(*self*)

Return height of bottom window in pixels.

Return Value

Size of bottom window.
(type=int)

initialize(*self*, *topPane*)

Inizialize Splitter. Decide wether to show both (top and bottom) panes, or only the first one.

Parameters

`topPane`: The top pane.
(type=wx.Pane)

setBottomWindowHeight(*self*, *height*)

Set the height of the Bottom Window and unsplit if new bottomWindowHeight is <= 10

Parameters

`height`: Height of the bottom window.
(type=int)

split(*self*)

Split top and bottom pane of splitter window. Sets the config.xml Var and checks the options-menubar entry

unSplit(*self*)

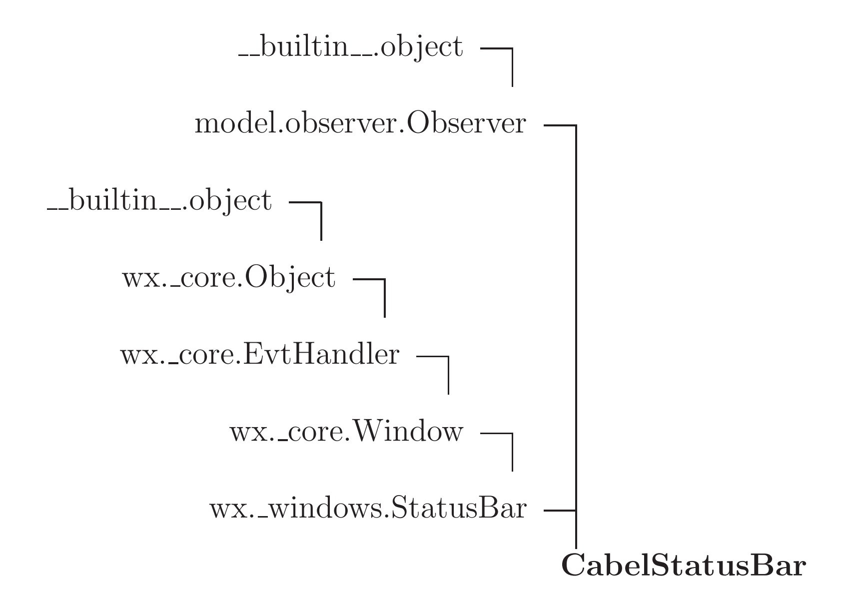
Unsplit. Only self._topPane visible. Sets the config.xml Var and unchecks the options-menubar entry

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __setattr__, __str__

A.16.5.2 Class Variables

Name	Description
MAX_HEIGHT_RELATIVE	**Value:** `0.75` *(type=`float`)*

A.16.6 Class CabelStatusBar



Cabel's own StatusBar.

A.16.6.1 Methods

__init__(*self*, *parent*)

Standardconstructor.

Parameters

`parent`: Parent frame.
(type=view.workspace.CabelFrame)

Overrides: wx._windows.StatusBar.__init__

onAutoplayCheckBox(*self*, *event*)

onPlayStopButton(*self*, *event*)

onSize(*self*, *event*)

onZoomEntered(*self*, *event*)

update(*self*, *observable*, *arg*)

This method is called whenever the observed object is changed. An application calls an observable object's notifyObservers method to have all the object's observers notified of the change.

Parameters

`observable`: The observable object.
(type=Observable)

`arg`: An argument passed to the notifyObservers method.
(type=object)

Overrides: model.observer.Observer.update extit(inherited documentation)

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __setattr__, __str__

A.16.6.2 Instance Variables

Name	Description
workspace	View workspace. *(type=view.workspace.CabelFrame)*

Anhang B

Weiterführende Links zu Csound

- Offzielle Csound Homepage
  `http://csounds.com`
- Csound Project bei Sourceforge.net
  `http://csound.sourceforge.net`
- Csound Benutzerhandbuch und Befehlsreferenz
  `http://www.csounds.com/manual`
- Csound Mailing Listen
  `http://www.csounds.com/list/index.html`
- User-Defined Opcode Database (Basis für Cabel Module)
  `http://www.csounds.com/udo`
- Csound Journal (ein internetbasiertes Csound Magazin)
  `http://www.csounds.com/journal`
- Csound Tutorials
  - Das 1. Kapitel des Csound Buchs
    `http://www.csounds.com/chapter1/index.html`
  - TOOTorials by Dr. Richard Boulanger
    `http://www.csounds.com/toots/index.html`
  - Mastering Csound by Dr. Richard Boulanger
    `http://www.csounds.com/mastering/index.html`
- Kompositionsumgebung blue
  `http://www.csounds.com/stevenyi/blue/index.html`

Literaturverzeichnis

[Anw00] Florian Anwander. *Synthesizer*. PPV Presse Project Verlags GmbH, 2000.

[Bou00] Dr. Richard Boulanger, editor. *The Csound Book*. The MIT Press, 2000.

[Pil04] Mark Pilgrim. *Dive into Python*. APress, 2004. `http://diveintopython.org`.

[uTAJ97] Charles Dodge und Thomas A. Jerse. *Computer Music – Synthesis, Composition, and Performance*. Schirmer, Thomson Learning, second edition, 1997.